Grammar-Oriented Object Design: Towards Dynamically Reconfigurable Business and Software Architecture For On-demand Computing

Ali Arsanjani

A thesis submitted in partial fulfillment of
the requirements for the degree of Doctor
of Philosophy

De Montfort University

2003

ABSTRACT

Grammar-oriented Object Design was shown to be a potent combination of extending methods, incorporating DSLs from a given business domain (BDSLs) and Variation-oriented Design in order to provide a seamless transition from business models to component-based software architectures. GOOD starts by extending current object modeling techniques to include the discovery and explicit modeling of higher levels of reuse, starting from subsystems, defining their *manners* using a domain-specific business language, i.e., using *use-case grammars*, that describe the rules governing the creation, dynamic configuration and collaboration of large-grained, business-process-scale, adaptive software components with pluggable behavior, through the application of architectural patterns and representation of component manners in the BDSL. This presents immense potential for applications in the domains of grid services, services on demand and a utility-based model of computing where a business need initiates the convergence of application components based on/from the manners of services they provide and require.

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

ACKNOWLEDGMENTS

I would like to acknowledge and express my deep appreciation for those who have introduced, led, mentored and shared with me the Path of Knowledge. "Knowledge is structured in consciousness" and the increase of the depth of one's knowledge is inextricably tied to one's personal evolution towards proximity to the Divine; Enlightenment, or knowing the Self. From the Greek "Know thyself" to the Vedic "Tat tvam asi" (That thou Art) to the precept that "he who knows Himself can only know God". My family blessed me with "philo-sophia," the Love of Knowledge: my Grandfather, Mr. B. Daftari, gave me not only knowledge of life but the wisdom of evolution which is more valuable; he instilled order, greatness and perfection; my Grandmother, Mrs. E. Daftari gave me the appreciation for the knowledge of social interaction and appreciation for the subtle, delicate aspects of life and knowledge. My Mother, Dr. Maryam Daftari, who has continued to inspire, support and guide me in the Path of Knowledge, is continuously unfolding wisdom for me in the Path of Knowledge and showed me the Dao of Knowledge. And then when one becomes a parent, God grants another kind of knowledge; one of the outward looking in rather than the child who looks outward from inside his own little world. My Wife, Parastoo, and Son, Sam, have been sources of support and love which equips one for the Way of Knowledge and have taught the delicate balance between tenderness and decisiveness.

To live 200% of life, the material must be instilled with the spiritual that transcends mere religion, but the practical application of the quest for Truth. This guidance comes from those who have achieved it, exude it, teach it and live it. Maharishi Mahesh Yogi is a living embodiment of the ancient tradition of the Masters of Vedic Wisdom which teach the direct realization of Knowledge, Pure Knowledge through the direct experience of the Self and of the divine, the source of Knowledge; Ved , pure knowledge.

As the Persian poet and mystic Saadi said, "Man can attain a state in which to whichever direction he turns, he sees God". Thus everywhere one turns, the signs of the Grace, Compassionate and Mercy of the Divine is there. The knowledge of God comes through knowledge of the Self and to the seeker comes Grace. He is the giver of Abundance and Merciful Grace; the All-knowing. To Him belongs all gratitude and appreciation.

But along the Way, many teachers have I encountered; of the more salient are: Dr. Mohsen Rostami, Dr. Hassan Mirian, Dr. Hussein Zedan. Dr. Zedan has been a great mentor and supervisor which I thank especially; he has been superb in his support of making even the most difficult aspect of knowledge look easy. His keen insight into issues technical and not have been a guiding light. I would like to also thank Dr. James Alpigini for the initial guidance on the thesis.

SUMMARY OF PUBLICATIONS

[1] Arsanjani, A., Grammar-oriented Object Design: Creating adaptive collaborations and dynamic configurations with self-describing components and services, Proceedings of TOOLS 2001, IEEE Computer Society Press.

[2] Arsanjani, A., Alpigini., J., "Using Grammar-oriented Object Design to Seamlessly Map Business Models to Software Architectures", Proceedings of the IASTED 2001 conference on Modeling and Simulation, Pittsburgh, PA, 2001.

[3] Arsanjani, A., "Patterns of Symmetry in Software Architecture," Proceedings of the Pattern Languages of Program Design, 2001.

[4] Arsanjani, A., "GOOD: Grammar-oriented Object design", Position Paper for OOPSLA Workshop on Metadata and Active Object Models, 1998, Vancouver, British Columbia.

[5] Arsanjani, A., "Dynamic Configuration and Collaboration of Components with Self - description," position paper submitted to ECOOP 2001 Workshop on Active Object Models and Meta-modeling, 2001.

[6] Arsanjani, A., Enterprise Component: A Compound Pattern for Building Component Architectures, Proceedings of TOOLS 2001, IEEE Computer Society Press.

[7] Arsanjani, A., "Rule Object: A Pattern Language for Flexible Modeling and Construction of Business Rules," Washington University Technical Report number: wucs-00-29, Proceedings of the Pattern Languages of Program Design, 2000.

[8] Arsanjani, A., "Analysis and Design of Distributed Java Business Frameworks using Domain Patterns", Proceedings of TOOLS '99, IEEE Computer Society Press 1999, pp. 490-500.

[9] Arsanjani, A., "CBDi : A Pattern Language for Component-based Development and Integration", European Conference on Pattern Languages of Programming, 2001.

[10] Arsanjani, Enterprise Component: A Compound Pattern for Building Component Architectures, Proceedings of TOOLS 2001.

[11] Arsanjani, Grammar-oriented Object Design: Creating adaptive collaborations and dynamic configurations with self-describing components and services, Proceedings of TOOLS 2001.

[12] Arsanjani, A., Alpigini, J., "Business Components have manners: Beyond Contracts", Submitted to the TOOLS Conference 2002.

[13] Arsanjani, A., Zedan, H., Alpigini, J., EXTERNALIZING COMPONENT MANNERS TO ACHIEVE GREATER MAINTAINABILITY THROUGH A HIGHLY RE-CONFIGURABLE ARCHITECTURAL STYLE, International Conference on Software Maintenance, 2002.

[14] Arsanjani, A., Levi, K., "A Goal-oriented Approach to Component Identification and Specification", Communications of the ACM, Oct 2002.

[15] Arsanjani, A., Hailpern, B., Martin, J., Tarr, P., Web Services: Promises and Compromises, ACM Queue, March 2003.

[16] Arsanjani, A., Explicit Representation of Service Semantics: Towards Automated Composition through a Dynamically Re-configurable Architectural Style for On Demand Computing, Proceedings of the First International Conference on Web Services, 2003.

PROFESSIONAL CONFERENCES

[17] Arsanjani, A., "Using XML to Perform Business-to-business Transactions", *Proceedings of XML World 2000.*

[18] OOPSLA Workshop on Business Rules Patterns I

[19] OOPSLA Workshop on Business Rules Patterns II

[20] OOPSLA Workshop on Business Rules Patterns III

[21] TOOLS Workshop on Component-based Software Engineering

[22] OOPSLA Adaptive Object Models I

[23] OOPSLA Adaptive Object Models II

[24] Software Development West 2000

[25] Software Development West 2001

[26] Software Development West 2002

[27] Pattern Languages of Programming 1999

[28] Pattern Languages of Programming 2000

[29] Pattern Languages of Programming 2001

[30] Pattern Languages of Programming 2002

[31] Patterns for Legacy Transformation, PLOP 2003-05-04

[32] Patterns for Building Dynamically Configurable Software Architecture Web Services, 2003, Pattern Languages of Programming.

Legend:
ACM - Association for Computing Machinery
TOOLS - Technology of Object-oriented Languages and Systems
PLOP - Pattern Languages of Programming

SUMMARY OF PRIMARY CONTRIBUTIONS

The primary contributions of this thesis can be summarized as follows:

	Term	Type (concept, technique, method, pattern)	Description
1.	Grammar-oriented Object Design (GOOD)	Method, Notion	An evolution of software engineering methods for creating a dynamically re-configurable architectural style based on enterprise components and services. An integration of component-based, service-oriented engineering with domain-specific languages, software architecture, business rules, self-integration of context-aware autonomous components.
2.	Business Grammar	Technique	A grammar representing the flow, structure and composition of a business domain.
3.	Variation-oriented Analysis and Design (VOA/VOD)	Method	A set of 6 techniques and artifacts along with examples to partition a system or part thereof into its more stable and less stable aspects.
4.	Axes of variation	Technique	Within VOAD, identifying, handling and externalizing axes of variation for greater stability in the maintenance of software applications and architecture.
5.	Subsystem Analysis	Activity in Method	Identifies key subsystems within the business domain and provides a formal description for them
6.	Goal-oriented Component identification and Specification	Activity in Method	Identification and specification of candidate component abstractions during business analysis based on the notion of encapsulating design decisions.
7.	Reuse levels	Concept and Activity in Method	Identify and Select reuse level
8.	Manners	Notion, Abstraction	Rules governing behavior of a component within a given context. Presented with a set of patterns for its implementation.
9.	Context-aware Software Component	Taxonomy/Type of component,	Defines the characteristics necessary to design and implement context-awareness of software components.

10.	Enterprise Component	Pattern	Provides a standard and consistent way of designing large-grained components in enterprise content.
11.	CBDi Pattern Language	Pattern Language	A Pattern Language for Component-based Development and Integration
12.	Rule Pattern Language	Pattern Language	A Pattern Language for Business Rules Modeling, Design and Implementation.
13.	Patterns for Web Services Architecture	Pattern Language	A Pattern Language for the design of service-oriented architectures.
14.	Dynamic Configuration, Collaboration	Notion and technique	Used to produce just-in-time integrating, on-demand application component assembly
15.	Mapping Business Architecture to Software Architecture	Notion and methodology	Defining a set of steps and artifacts that result in the more seamless mapping of business architecture to component-based software architecture. This helps bridge the semantic gap between business and I/T through the creation of an abstract formal specification of the high-level business functionality that can be mapped directly onto component-base software architecture.
16.	Extensions to current methods for component-based and service-oriented software engineering	Methodology, techniques, workproducts, activities	A set of activities and workproducts have been identified and utilized on projects since 1994 that extend object-oriented methods for component-based development. Examples:
17.	Dynamically re-configurable architectural style	Blueprints and reference architectures	The blueprint (components, connectors and constraints) for creating an architecture that will support dynamic re-configuration and re-composition

1 Chapter One: Issues in the Design and Implementation of Software Architecture and Applications

Objectives

- To discuss the issues and problems prevalent in the design and implementation of software architecture and applications based on the literature and field experience; describe the need to solve a set of related problems.

1.1 THE PROBLEM: THE GAP BETWEEN BUSINESS AND I/T

Businesses lose revenue, time, budget and reputation in failing to stay within the constraints of their often rapidly changing functional and non-functional requirements; imposed through a variety of "forces" such as legislation, market pressure, deregulation, competition, customer trends and business policies. The challenge is to remain within the allotted time within the set budget and yet have the ability to manage changes to requirements and implement the new changes in a timely and cost-effective fashion.

These forces often include total cost of ownership; lowering currently high maintenance costs, having the ability to rapidly introduce new products and services into the marketplace, tap into legacy applications to recompose an application to handle new business goals and models.

Integration is found to be key in elevating an enterprise into higher levels of capability towards the ability to achieve the degree of resilience and responsiveness that is required in today's rapidly changing and on-demand world of interactions and transaction across vast value-nets.

The gap between business I/T is even more pronounced when we see the current state of I/T being at a level 0 in the following scale of levels of integration within the enterprise.

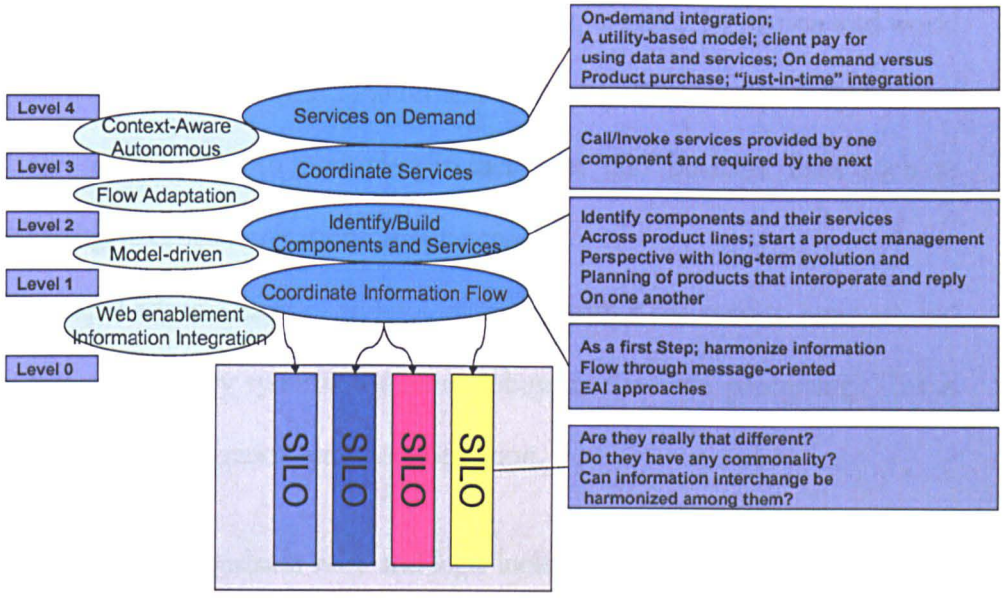


Figure 1: Levels of Integration

I/T systems are typically at level 0 while the expectations of the on-demand world of business are often at level 4. In order to bridge this gap we need to elevate the capabilities of I/T systems from level 0 to a higher level, gradually migrating key systems to level 4, although the prospect of taking all system to that level of maturity is not going to be operationally viable.

Integration (of data; EAI or enterprise application integration) is a good start as we go from level 0 to level 1. However, integration of processes is also required. This cannot be done in an ad hoc manner; therefore components must be identified and built or mined out of legacy systems that enable process integration within lines of business at level 2. Further, the exposure of services across the value-net and across lines of business in an interface-only fashion (as in Web services) is key to supporting business needs at level 3.

It is vital for a business to be able to adapt to variations in requirements, technology, business goals and processes as well as new yet slightly dissimilar software architectures that need to be created.

A Business has many challenges. Competitors, market, legislation and world politics often impose the requirements volatility. This is especially true for large businesses with multiple product lines and business lines such as banking, telecommunications, insurance, etc.

The rapid advancement of technology brings with it the need to renovate and adapt older legacy systems for more robust and reliable processing. This is called legacy integration and transformation.

In addition, the business rules and logic locked within often monolithic legacy systems is often required to be tapped into so, for example, web enabled systems with added value can not only enhance a business lines current functionality but also maintain the smoothness of its operation by tapping into the units of its legacy functionality.

Thus the primary factors of successful software architecture are summarized as:

- Component-based; reuse-driven
- Architecture-centric
- Aligned with business goals and direction
- Being able to re-configure and re-assemble parts to produce new applications

- Adaptive, flexible and changeable

- Integration or transformation of legacy systems into currently usable assets

Over the past 20 years of the author's experience in large scale systems development, experience has shown that these problems are best addressed when managed across five domains. The domains of component-based development and integration are explored in the following section.

1.1.1 Five Domains of Component-based Development and Integration

One of the first key impediments to component engineering has been the false assumption that components are primarily a technology and tooling issue. Although there are various technology and deployment-based solutions to creating fine-grained components, the full realization of component-based software engineering (CBSE) on a large scale ("programming-in-the-large" [138] or "megaprogramming" [92]) requires a focused set of solutions not only for the technology arena, but also in the organizational, business, and operational (e.g., project management) areas.

There are two fundamental questions to ask: "What (should) become a component? What are the boundaries based on which we should define components?" and "How do you build this component?" Both these are distinct, but related to "what is a component?" The selection of "what

becomes a component?" follows from the needs of the business domain. Components are first a business issue then a technology problem.

Addressing component-based development and integration issues in an enterprise context requires attention to five domains of concern where activities are done and artifacts produced. The *domains of CBDI* are defined as: organization, methodology, architecture, reusable technology implementation and infrastructure (see Table 1: How GOOD relates to other disciplines in Software Engineering). For each of these domains, we have identified and executed on producing workproducts and activities that support the aims of that specific domain as it pertains to enterprise scale component-based and service-oriented computing. This comprises the method proposed as part of GOOD which has been implemented on industrial strength projects.

Domain	Scope
Organizational/ Business	Reuse across the organization; reuse programs, notions of CBDI, knowledge transfer; management of CBDI projects, team structures, hand-off protocols; business sponsorship and visibility
Methodology/Process	Extensions to current methods (Unified Process and GS Method) to fully support CBDI from the ground-up (vertical support) and across the life-cycle (horizontal support)
Architecture	Patterns for model, design, build, test, deploy, and maintain component architectures in green-field and legacy integration/transformation tracks.
Reusable Technology Implementation	Best-practices and patterns on Mapping the architecture to an implementation technology (each is called a mapping): Enterprise Java Beans, Web Services, Java Servlets/Java Server Pages, CORBA, .NET
Infrastructure	Tooling; development, test, pre-production and production environments,

Table 1: The Domains of Component-based Development and Integration

1.2 Issues in the Design and Implementation of Software Architecture

The gap between the Business Domain within commercial organizations and Information Technology (I/T) is primarily a result of the conceptual gap between the tools, terminology, methods, models and the mismatched expectations that result from dichotomy. This gap is problematic in that software must clearly reflect, support and be aligned with the often rapidly changing needs of the business domain. Therefore a mapping from one domain to the other is required. This mapping is often done in an ad hoc fashion resulting in great risk and expenditure to projects. Where it is successful, it is often the result of valiant attempts of "heros", often not the consistent and repeatable result of a mature software industry.

Mapping Business Architecture to Component-based or Service-oriented Software Architecture has been an area that presents more promise in solving this problem, because components and services naturally form units of encapsulation that can be built with less ripple effects on the rest of the system once changes come dashing in from changes in business requirements.

However, this large gap still exists and the promise of brining business and technology closer together to achieve greater repeatable success on projects and on on-going maintenance has not been fulfilled as 80% of projects that

are started are cancelled and of that remaining 20%, 70% are over time and budget.

One of the main problems in this gap, is the inherent imprecision of natural language requirements specifications coming from the business domain that are handed over to I/T. Such specifications are open to risk-ridden interpretations that may not only miss the business goals and objectives but actually impede business development through lack of flexibility in resiliently accommodating new requirements. This is one of the major issues in software development today.

Although there are many issues and problems that are encountered within the software development life-cycle, the current chapter attempts to capture the more salient ones based on the author's experience and based on industry literature.

The context in which these issues and problems arise are in the design and implementation of software architecture and applications on large-scale enterprise-wide projects in larger organizations (such as banks, telecommunications companies, financial institutions, healthcare, etc.). They have been found to primarily consist of issues relating to : *incorrect selection of component granularity, treatment of component-based software engineering as object-oriented software engineering (objects only as fine-grained components), lack of first-class and full support for components and services in current Object-oriented methods, lack of consideration of business rules as first –class constructs of the component paradigm and*

finally, the lack of the ability to rapidly and dynamically re-compose and re-configure a component assembly. Other issues have been documented in [14].

Not only are the tools and languages inadequate, but a deeper problem exists. The methods, process models and methodologies prevalent in object-oriented circles fail to adequately support the identification, specification and design of software components and services. And indeed they fail to take into account the notion of their *component context* and configuration within the face of changing requirements. A component context can be described with an analogy: hardware integrated circuits (ICs) are plugged into a silicon board, which provides the buses and connections between elements on the board. The software components are plugged into a similar domain, architecture and application context, which provides the practical constraints on the collaborations and interactions between components and services. Components are mentioned in the context of a component-based development and integration context and services are related to service-oriented architectures in particular the variety implemented as Web Services.

1.2.1 Granularity

Components granularity is often tacitly assumed to be too fine-grained, creating an object graph with high degrees of dependencies making reuse, Refactoring and re-composition difficult if not impossible.

The first issue under consideration is that of *Component Granularity.* The scope and context of the discussion is at the enterprise scale that often deals with families of applications; product lines and business lines. This requires

large applications that interact to support the business within the entire enterprise. To this end, it is convenient to define three levels of component granularity: fine-grained "traditional" objects, medium-grained business subsystems (pricing engine, shopping cart subsystem) and large-grained semi-applications (billing system, account management system) that are customized and composed to produce enterprise scale systems (e.g., internet order-processing system).

Components are most often discussed and handled within the same context as if dealing with objects in the object-oriented paradigm. Object-oriented projects tend to result in rather convoluted graphs of inter-dependent object. The entropy introduced by creating these unmanageable object graphs arise from the perspective of using a large number of fine-grained objects (here used in a general sense and held to be – in this context— synonymous with the notion of class).

It is noteworthy that the author's experience points to the fact that large enterprise object models seldom find their way into working systems; agreement across business lines within an organization is difficult to achieve. Thus, an Account class may have different connotations for each product line or business line: "This is not my Account semantics." Starting from the axiom of fine-grained classes or objects often results in a design that is found to be weaving a web of (often unnecessary and impractically limiting

set of) dependencies (within the context of technology) and lack of consensus (within a business context).

Whereas this paradigm is certainly applicable on smaller-scale projects of the order of a few developers with a few dozen classes, large enterprise systems require a larger granularity of partitioning and separation of concerns. This can be done through the explicit selection of reuse levels [8] . A "class" is the finest-grained (lowest) level of reuse. Recursively larger grained, named, clusters of classes are required to partition and manage the conceptual and operational complexity of large numbers of classes. This problem is similar to that encountered in state diagrams where there is a combinatorial explosion of states. The solution, introduced by Harel is to use a modularized hierarchy of states, represented as statecharts. Similarly, we propose that medium- to large-grained components recursively aggregate smaller-grained classes into larger modular constructs that can be manipulated and managed as a single entity using key attributes and capabilities as outlined in the Enterprise Component architectural pattern (see Figure 2) which defines the set of underlying patterns for designing large-grained components.

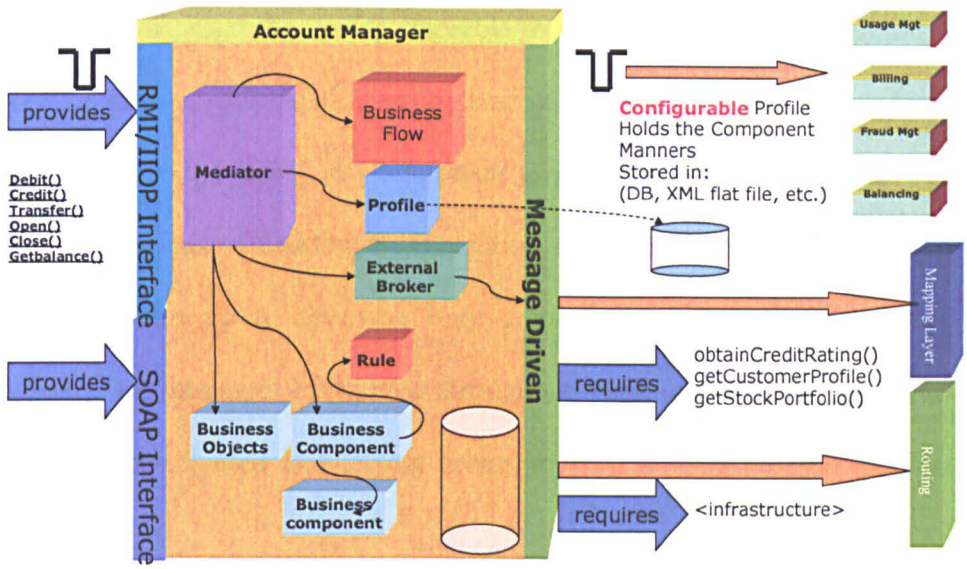


Figure 2: Enterprise Component Pattern

It must be recognized, however, that the design and maintenance of medium- to large-grained components requires a different mindset. This mindset is based on deciding on the parameters for separation of concerns and of the design decisions that need to be encapsulated by the choice of component boundaries. In addition, this requires more mature perspectives on not only the notion of reuse (as depicted in the concept of reuse levels) but of the attributes of pluggablity, interoperability, dynamic configuration, dynamic collaboration (dynamic choreography) and self-description.

As depicted by the concept of reuse levels, fine-grained objects are not the only level of granularity of reusable entities or components. Assuming that fine-grained objects are the unit of reuse is a fallacy that leads to large, mostly un-maintainable object graphs that tend to take on a design direction

of their own rather than being traceable to a business goal. By targeting a level of reuse appropriate for an application context, the issues arising from the improper addressing of Component Granularity can be mitigated. For example, instead of building an enterprise object model that business lines will not agree upon; choosing large grained components with services exposed and messages exchanged through choreographies in a component-base service-oriented architecture seems to yield a more realistic, feasible solution.

This constitutes the explicit acknowledgment of the existence of, subsequent identification of and appropriate selection of higher levels of reuse from the very outset of the project; starting at the beginning of the life-cycle in business modeling and analysis. As a contribution to business modeling and analysis we introduce the notion of Subsystem Analysis[1].

'Reuse Level Selection' is the first step of subsystem analysis. Then, the domain is decomposed into a set of subsystems based on the selected level of granularity. Usually, such subsystems are Composite (in the sense of design patterns). Yet current methods do not support the explicit decision-making on what level of reuse to set as the unit of reusable granularity for a project. For example, the spectrum of reuse levels spans individual base classes all the way to subsystems (collections of classes).

---

[1] See chapter on Extensions to Current Methods for Component-based Software Engineering, Subsystem Analysis.

Granularity can apply to both components and services. Figure 3 shows that the decision on granularity impacts the size, scope, usage complexity and performance of the software components and services.

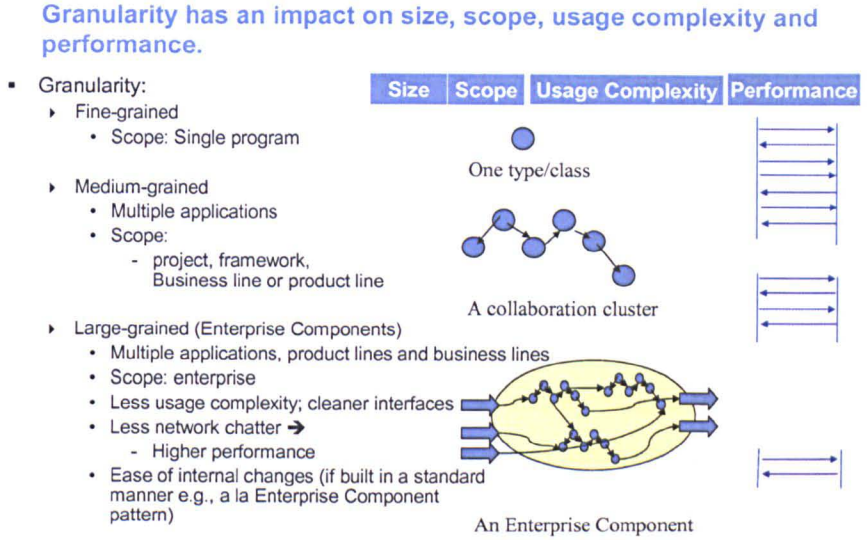


Figure 3: Granularity of Components and Services

Appropriate granularity naturally comes when a correct method is applied to the discovery of component boundaries. This contribution of the steps in using GOOD from a method extension perspective is described in detail in Chapter Nine, 7.1 Extensions to Current Methods for Component-based Software Engineering.

1.2.2 Semantics of Component Composition and Collaboration (aka Manners)

Although interfaces and in some rare cases, contracts are indeed used in software development, how the contract is to be designed (not implemented) is not specified. Although from a consumer of the contract the "how" (or abstract specification of component context-sensitive behavior) should be transparent, from the perspective of the component designer, the provider of the services exposed by the component, this specification is critical. Without it systems are designed in inconsistent and non-standard ways, policies, rules and contextual aspects of the component, its composition and collaboration are left to be hard-coded. This tight coupling and implementation dependent nature of the policies, rules and context make changes to the component, its composition and collaboration framework difficult, if not impossible to manage in a cost effective and timely manner.

Protocols have been offered as partial solutions to this dilemma but fail to solve the problem. The way in which groups of methods on an object or functions in a structured module are to be used; the sequence of usage and dependency structure (which one is called before another one) is left to ad hoc hard-coding rather than a declarative specification that facilitates maintenance through the ability to be dynamically re-composed or re-configured.

To address this problem Grammar-oriented Object Design (GOOD) introduces the notion of manners described in detail in Chapter Four: Manners.

1.2.3 Gap between Business and IT Architecture

This gap contains several detailed aspects, each of which deserve their own attention. Various aspects of this major problem in software engineering are described below.

1.2.4 (Non) Evolutionary Cohesion

Non-cohesive, divergently evolved architectural elements. Consolidate or "assimilate" disparate processing that is the natural result of organic and unstructured architectural growth of information systems. This calls for removing redundant functionality across multiple heterogeneous systems, evolved over time with differing business goals architectural discipline and infrastructure. Sometimes this consolidation occurs when a business grows and its legacy systems can no longer withstand its volume, throughput or scalability requirements; not to mention the need to have drastic additions to its functionality.

1.2.5 (Lack of)Adaptation to Change

Functionality changes or software maintenance accounts for over 60% of software development costs and sometimes more than 70% of the project effort [21].

1.2.6 (Lack of) Business Traceability

Lack of traceability and business goal and rule centralization. Often there is no central repository of business rules that can be used as a frame of reference to trace back the functionality of software components.

1.2.7 (Lack of Appropriate) Modularization

Lack of proper modularity or componentization. Thus with a lack of traceability the need invariably arises to replace obsolete rules and "plug-in" new ones in a modular and componentized fashion.

Lack of configurability. Systems are built five times over rather than building one system and configuring it five times. This often occurs within the context of product lines or families of applications that have similar but sufficiently varying needs.

1.2.8 (Lack of consideration of multiple) Reuse Levels

Reuse is an elusive goal. It has been the holy grail of software engineering for many decades. One of the major issues is that the granularity of reuse impacts the type of reusable element being designed and implemented and later maintained. Reuse is thus not only a technical problem but a very large and deep-rooted organizational issue [48]. One of the key fallacies in software development projects is the primary reuse of the notion of class or object. Contrary to this, real reuse is achieved and is achievable through the consideration of higher orders of reuse. The selection of this level of reuse sets an achievable objective for the project and does not waste time in focusing on what is not of primary importance within the context of reuse.

Ten levels *of reuse have been identified* [8]. They are, in order of increasing abstraction: base class, aggregation hierarchy, inheritance hierarchy, and

cluster, i.e., subsystem, framework, component, pattern, generic architecture, environment interaction meta-knowledge and technology transfer knowledge. The base class, i.e., level zero, is often the tacitly assumed reuse level that projects employ, leading to fine-grained objects that build "forests of inter-wined object graphs" which are difficult, if not impossible, to reuse in part. Our experience and research indicates that higher levels of reuse should be chosen for designing reusable components by modeling subsystems--at design-time, and realizing them as components at run-time. Methods have typically emphasized the run-time aspects of components; e.g., the UML's icon for a component provides no guidance on what the steps are to realize one: where to start design and what to include in a component.

1.2.9 Reuse

Lack of reuse. Writing reusable software requires more than the valiant efforts of a few developers; it requires a reuse program and full organizational support. However, this approach [48], as not been successful in the large. Although there have been instances of success, a repeatable pattern has been difficult to achieve. Many reasons exist [18]. A salient one being the fact that reuse is not tackled from the five domains of component-based development and integration [2][4].

1.2.10 Flow Variation and Adaptation: Applying Changes Non-intrusively

Changes to software architecture and applications are often applied intrusively resulting in long testing and validation cycles that often exceed the tolerance of the timeliness required supporting business goals.

Change is not anticipated explicitly in the methodologies employed in software design and construction. Grammar-oriented Object Design presents a method (see Chapter Nine) that is an extension to current methods such as Rational Unified Process, IBM Global Services Method, Method /1, etc. The extensions fill a need for the first-class support of components starting from business analysis down to deployment.

This section contains an elaboration of the requirements for applying changes non-intrusively.

1.2.11 An Analogy for context and implications of flow variation and adaptation

Within this section, an analogy from the domain of theoretical physics will be used to provide a richer description of the context and implications of the current theme on flow variation and adaptation.

In 1905, Albert Einstein wrote a paper entitled "On the Electrodynamics of Moving Bodies" in which he introduced the special theory of relativity. In

this he proposed that bodies whose motion approached the speed of light were *governed by a different set of laws* than those traveling at lower velocities.

Another challenge is the increasing demand for *higher Configurability and dynamic re-configuration.* Configurability refers to the assembly of an application within a family of applications from parts that are enterprise components that can be dynamically re-configured and re-wired into a working application; often designed to meet a volatile set of functional in a moving landscape of non-functional requirements.

In modern day computing, software objects move through the implementation of business specifications, or more accurately, move through versions and configurations based on changes in their requirements. Here, requirements are the frames of reference for software objects.

Applying changes to software architectures to meet a continuously changing landscape of business requirements and technology issues and constraints is a major challenge of modern software engineering, akin in analogy to the changes in the frame of reference of Newtonian Mechanics where the speed of light was shown to be a limit and the velocities approaching that of light required a different set of equations to describe natural phenomena for moving bodies.

Although object technology has provided great advances in the way we conduct the software engineering process, it has not fulfilled many of the goals of large-scale reusability, ability to rapidly introduce variations (slight changes, not those requiring major restructuring) to (often running) software.

Component-based software engineering [25] is an evolution of the object paradigm. Service-oriented architectures are a wrapping of services and capabilities around a component infrastructure. However, realizing CBSE has had its challenges primarily due to a large myopic focus on technology-related aspects.

Recognizing that components are a means to an end of highly re-configurable and adaptive architectures that are able to shadow the changes in a business domain with ease is a step in the right direction.

Recognizing that five domains of CBSE must be simultaneously addressed [10] is key to realizing "On Describing the Variations of Changing Requirements" to extend Einstein's metaphor.

Adaptive Architectures [101] have been suggested to address some of these issues but are constrained to more structural types of adaptation than being able to dynamically change business rules and business flow.

Patterns and pattern languages have been proposed [9] to solve commonly encountered problems. These are based on a wide variety of project experiences applied across multiple industry segments. These help in defining standards and best-practices for building large-grained, message-aware enterprise business components that can be used to build applications that function across multiple product lines and business lines.

1.2.12 Current Methods do not explicitly Design for Change: Change and Variation-oriented Design (VOD)

An examination of current methodologies shows that they do not have explicit support (other than, for example, the variations in a use-cases or "change cases") for designing for change and variability of the software architecture and applications that will be running on it. GOOD provides explicit support within methods (see chapter nine) in the form of Variation-oriented Analysis and Design, employing externalization to achieve resilience in the face of change [13].

VOD was first mentioned in [5] by Gamma et al. Arsanjani expands VOD into a discipline with a set of explicit methodological steps and techniques. The author introduces Variation-oriented Analysis (VOA) and develops six principles of VOAD. This extends VOAD to the notion of dynamic non-intrusive re-configuration [5][11] where dynamic configuration,

collaboration and self-description of the component services and component composition into applications are realized.

Variation-oriented analysis and design is based on separating out the changing from the less-changing, more stable and permanent aspects of a business domain, software architecture and finally, detailed design at the class level. The latter is the level of granularity mentioned by Gamma et al. in their seminal Design Patterns work.

Arsanjani further extends the notion of designing for variations, a notion that can be called "Design for Change" relying on and expanding on concepts such as Meyer's Open-Closed Principle [66].

1.2.13 Types of Change

In this section we will explore the impact of changes to code (existing systems) from several aspects: life-cycle, intrusiveness, unit of composition /encapsulation, and timing. These aspects have been empirically experienced as critical to change by the author or many years on multiple projects.

1.2.13.1 Life-cycle

Changes can occur all through the software development life-cycle; from requirements all the way down to implementation and deployment. As the changes are made to finer grained workproducts of the development life-cycle, such as detailed design or even worse, to code, the resulting modifications are increasingly costly.

Let us focus for a moment on the changes to existing systems which is in line with many of the problems encountered in actual software engineering projects in industry.

1.2.13.2 Intrusiveness and Unit of Composition

Changes to code can generally be categorized as being

- Intrusive, or
- Non-intrusive

With respect to a unit of composition or encapsulation (generally a reuse level; but more specifically, we will focus on the following more commonly encountered units)

- method,
- object, (class)
- cluster (of classes; group of collaborating objects),
- finer-grained component,
- framework, (medium-grained component)
- large-grained component (enterprise component) or
- Subsystem (containing several enterprise components).

1.2.13.3 Usage Time

These types of change and unit of composition can be applied at various points in the life-cycle of a component composition:

- Design Time
- Usage Time
- Assembly or composition Time
- Start-up
- Configuration
- Run-time

Therefore, the objective of VOD is to minimize the cost of making changes with respect to the above aspects of a software system, namely phase in the life-cycle, unit of composition, usage time (timing) and intrusiveness.

1.2.14 Methods and Methodologies

The third flaw of **current component-engineering methods** is that object-oriented analysis and design methods do not fully support the design and construction of medium- to large-grained business components *across the development life-cycle: from business modeling to deployment and maintenance.* If we agree to enforce Large Component Granularity then we must augment software engineering methods to support the design of larger-grained components. This calls for Subsystem Discovery (versus Object Discovery) and Subsystem Analysis subsequent to Domain Analysis. Subsequent steps in this process is discussed in the last chapter of this thesis.

The **lack of methodology support** for component-based development from a holistic business-focused perspective has detrimental effects on the

software life-cycle. Current methods do not provide direct upfront, leading life-cycle support for business architecture design and modeling leading to component identification and specification.

Having attempted to address the same basic problem or reuse and configurability, **Object-oriented methods and programming have essentially fallen short** of their expected progress in this area. Systems continue to be hard to reuse and difficult to maintain. Smaller grained components or objects (here used to cover the commonality between the stricter delineation of classes, interfaces, and types) tend to create large cluster or object graphs with high dependencies in terms of inheritance, composition and reference to other objects. Often, reusing one object or cluster entails using its dependents and those it depends on as well, defeating the purpose of light-weight reuse. The lack of success of many large small object-oriented frameworks, despite an elegant design is a testimony to this (e.g., San Francisco Project, Taligent's Framework).

Configurability, symmetry and stability. Component-based development [25], an evolved form of object-oriented development, has been attempting to solve the same problem through a black box composition metaphor of ports, connectors and interfaces. More recently, notions of aspect-oriented programming [98], subject-oriented programming [98] and multi-dimensional separation of concerns [98 (251)] have been introduced to deal

with increasingly overwhelming issues in this space. Product line architectures concentrate on building components that can be reused across a family of applications [51][57][58].

Even with this impressive landscape, we are still faced with the same basic questions of reuse, configurability, customization, stability and adaptation. Within this sphere of change in architecture, we must note that hard-wired connections between components in an architecture are based on (often) tacit assumptions about the non-changing aspect of the connectors or components. Entropy is the amount of disorganization that creeps in as the result of the third law of thermodynamics as applied to physical systems. The same notion, applied to cybernetics via the work of Norbert Wiener, suggests that entropy can also be measured within information systems. Specifically, the entropy of a software architecture increases with every change made to its structure and function; because the intention that the software was to change along that axis of variation may not have been present as an architectural design decision when the system was designed. Focusing on the goals and intentions that the business has set out to achieve, even on a per-iteration basis is critical to its success and the success of supporting software.

1.3 The Solution: dynamically Re-configurable Architecture through grammar-oriented object design

1.3.1 Motivation

To achieve the above, the author proposes *Grammar-oriented object design* (GOOD) as a new way to solve "old" problems in software architecture. It is the application of subsystem analysis; variation-oriented analysis and design using domain-specific languages that help define an enterprise-scale, loosely coupled, business-driven component architecture. This allows the creation of a new architectural style that has the attributes of dynamic re-configurability and re-composition based on changing business requirements and I/T non-functional requirements. This architectural style is composed of medium- to large-scale business components, services and their interfaces, contracts and manners in a set of architectural layers mapping to an n-tier architecture.

Rather than emphasizing the components within the architecture, this style focuses on the more important problem of dynamic composition of enterprise components and services that are guaranteed to support business processes and goals.

Grammar-oriented object design is the next step in the evolution of software engineering methods. From procedural programming and design to object-based and object-oriented programming and then on to object-oriented design and analysis (in the order of their evolution), we next arrive at

component-based software engineering and recently to service-oriented software engineering. The notion of subject-oriented or aspect-oriented programming paradigms [41] is powerful metaphors which are based on the multi-dimensional separation of concerns. Object-orientation is based on the separation of concerns across one dimension.

The next step needs to fulfill the following goals which can be summarized by the name "on-demand computing":

1. Dynamic configuration
2. Dynamic collaboration
3. Self-description of semi-autonomous components
4. Just-in-time integration or rapid component or service assembly based on business goals and technical aspects (often relating to non-functional considerations and requirements)
    a. This is also referred to as on-demand computing.
    b. One of the corollaries of this approach is a utility-based model of computing

1.4 THESIS STRUCTURE

Therefore, what is needed is an integrated methodology that enjoys the combination of component-based, service-oriented software engineering, domain-specific languages and software architecture.

In chapter one commonly encountered problems and issues in the design and implementation of software architecture are described in detail. This lays the foundation for many of the problems that GOOD helps solve or alleviate to a greater degree. Chapter two is a literature review which discusses the evolution of the notion of software components. Subsequently, in Chapter Three the solution to the problems detailed in Chapter One is shown to be solved through the introduction of Grammar-oriented Object Design. Chapter Three describes one of the main contributions of this dissertation, namely Grammar-oriented Object Design (GOOD) in terms of semi-formal, re-configurable, executable business specifications. The subsequent two chapters (four and five) elaborate the details of the novel contributions of GOOD by describing Manners and Executable Business Specifications. Chapter Six describes the impact of GOOD on the disciplines of software engineering such as Business Architecture, Software Architecture, Service-oriented Computing and On-demand computing. In Chapter six, we concentrate on the notion of software architecture and its relevance to our discussion within the context of architectural styles that can support the needs of dynamic re-configuration or rapidly changing business needs. Chapter six also explores

the implications of GOOD for service-oriented architecture and web services on demand computing where the leveraging of open standards and enablement of just-in-time integration paradigms is described using the contributions of this thesis. Chapter six also explores the domain of business modeling and business architecture to provide a background for the reader as to what is missing in this domain and how this dissertation contributes towards filling that gap.

Chapter seven describes the method contributed by Grammar-oriented object design that creates a dynamically reconfigurable architecture based on a service-oriented architecture. This chapter introduces a set of process steps in the context of the GOOD methodology for developing service-oriented, component-based systems that exhibit dynamically re-configurable characteristics. This creates the platform for on-demand computing.

Chapter eight is an evaluation of the contributions by demonstrating a canonical example using a case study called E-bazaar.

Finally the conclusion summarizes and concludes the dissertation. The structure is depicted in Figure 4 below.

The following diagram summarizes the structure of this dissertation:

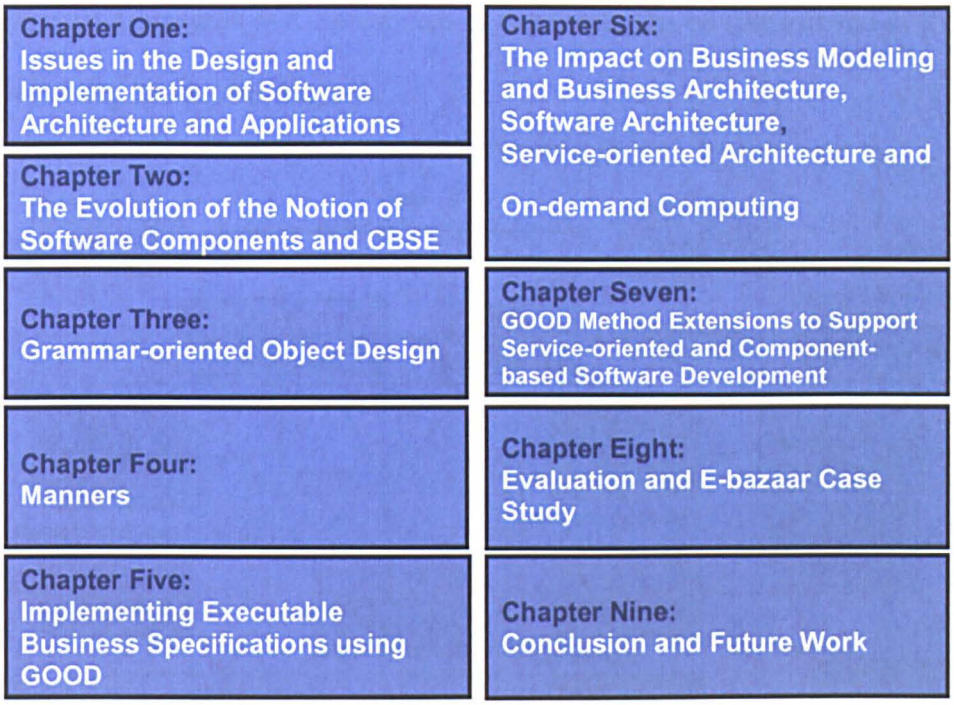


Figure 4: Structure of the Thesis

1.5 KEY CONCEPTS

The set of primary concepts can better be grasped by understanding their mutual relationships with one another. The main concepts and notions discussed, analyzed and contributed in this thesis can be summarized in the following conceptual diagram.

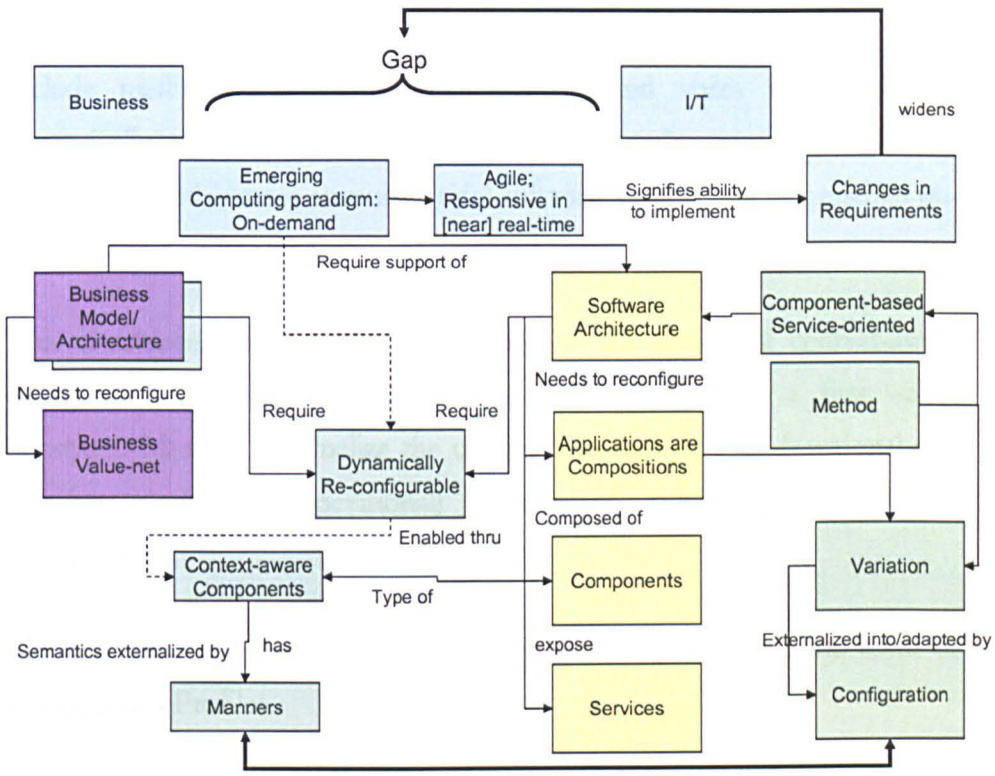


Figure 5: Meta-model of Concepts in Thesis

The gap between business and I/T is shown above and seen to be aggravated or widened as changes in requirements come about making changes to typically brittle I/T systems difficult to implement in a timely fashion that would keep up with the business domain. Therefore, to enable the emerging paradigm of on-demand computing which allows the just-in-time horizontal integration of systems and businesses across a supply chain to respond to customer needs and outside threats, we require agility and the ability to respond in near real-time to these changes in requirements and new needs.

This agility and responsiveness requires a dynamically re-configurable business model/architecture and software architecture. The business model or architecture must be able to rapidly reconfigure its value-chain and the way it conducts business in that value chain. The software architecture has applications running on it which consist of components that encapsulate functionality offered as services.

The main contributions of this thesis are found in the green boxes which include methods and architectural patterns and styles for developing dynamically reconfigurable component-based (service provider side) and service-oriented (service consumer side) software architecture that enables on-demand computing.

This is achieved through the introduction of the notion of context-aware components and services (CACs) that have manners as a first –class construct. Manners externalize the composition and flow of functional and non-functional (i.e., operational) aspects of software and business architecture.

Manners are often externalized in Enterprise Components in the form of Configurable Profiles.

The variations that come about as a result of changes in requirements are dealt with and defined by the method which includes variation-oriented analysis and design. These variations are shown to be identified, separated, encapsulated and externalized within the manners of the system and its components.

2 Chapter Two: The Evolution of the Notion of Modularity, Software Components and Services

Objectives

- To describe the evolution of the notion of software components within the literature and relate them to the issues in chapter one
- To describe the requirements for software components and how they drove the evolution of the notion, techniques and methods for designing, creating and managing software components
- To explore the notions of granularity and composition of software components
- To identify the relationship between software architecture and software components
- To explore the relation of components with object-oriented types

2.1 The Evolution of the Notion of Modules, Software Components and Services

2.1.1 A Literature Review of the Junction Point

The topic of this dissertation stands at the junction point of several disciplines in computer science, most notably, software architecture, component-based software engineering, patterns and best-practices, domain-specific languages and architectures, formal specifications, business rules, business modeling and software development methods.

Software projects exist to support the business or application domain; they are not a *Ding-an-sich*[2]. These projects supporting a given business domain should consciously be initiated from a business driven perspective, construct a business architecture and then map that onto a component-first software architecture. The configuration or composition of components is infinitely more important and complex than the sum of its constituent parts. This composition or configuration however needs to be made configurable for different component contexts. This need gives rise to the need for a Dynamically Re-configurable Architectural (DyRec) Style.

This style lends itself to rapid business growth through lower maintenance costs through more configuration for adaptation than customization of a set

[2] "Thing-in-Itself". Immanuel Kant, Critique of Pure Reason.

of components, faster time to market through the enablement of agile introduction of products and services into an extremely competitive marketplace, and the introduction of product line spanning, enterprise components whose manners are typically externalized for ease of adaptation – even at runtime.

In order to expound this new discipline of software engineering, namely, Grammar-Oriented Object Design (GOOD) which focuses on the goals of the business and can thus also be called Goal-Oriented Object Design, we need to explore and understand the overlapping aspects, key interrelationships and seminal impacts of the above areas of software engineering amongst themselves.

We will start by what is now considered a classic; namely, David Lorge Parnas' 1972 paper "On the Criteria to Be Used in Decomposing a System into Modules" [72]. This acclaimed paper has been considered seminal to the creation of the object paradigm; primarily due to the introduction of the concept of information hiding. Upon closer study, however, the paper's main import and implications are deeper than the commonly perceived notion of information hiding. The paper's main theme is, rather, the more general *concept of encapsulation of design decisions*. This allows the changes to a software system to be localized rather than have wider ranging ripple effects. The emphasis is so much on understanding and isolating variations that we

call this notion "variation-oriented design" (VOD) [110]. The notion is not unfamiliar to the design patterns and domain engineering communities. Encapsulation of variations on design decisions is of key importance to introduce stability in software systems and thus software architecture. The notion of designing-in change points into a software structure has been explored most notably by Pree [75]. Fayad has explored the notion of software stability through the notion of Enduring Business Themes (EBT) [37]. However, EBT's do not handle or discuss the notion of variations and how to handle them in software application and architecture design.

One of the contributions of this thesis is the introduction of a set of patterns for software symmetry and stability [3], describing how to achieve stability and design for changeability and variations in applications and architecture. This higher architectural stability is gained through monitoring aspects of symmetry in software architecture.

Complementary to the notion of stability or less-changing aspects of software architecture is the notion of Variations. Parnas' paper [72] introduces the notion of information hiding, which was later the foundation for abstract data types and subsequently, the object paradigm. This principle is often enunciated as follows: "an abstract data type should conceal its data structure and implementation mechanisms from the outside world or clients

who invoke its services through a set of functions that are exposed and can be used to manipulate the underlying mechanisms of the data structure.

It is interesting to note that the frequently overlooked aspect of Parnas' paper is that design decisions about varying aspects of the algorithm should be encapsulated; information about the decision should be hidden (encapsulated) so that when we decide to change this decision the impact will be minimal.

This thesis, in Variation-oriented Analysis and Design, shows how hiding the information about the design decision or mechanism implies the encapsulation of *variation points*. We introduce six principles that describe the process of identifying these variation points and how to externalize them [14].

Each design decision is a potential variation point that once exposed, exposes the architecture to entropy with every change made to that variation point. Eventually, the architecture is lent brittle as the number of variation points, not having been separated out conceptually or physically from the rest of the design or code sustains changes to its implementation through new design decisions.

Thus, it is crucial to identify, reify and externalize [14] these variations points in software architecture, often done well through encapsulation within the context of an Enterprise Component [9].

We discuss the importance; propose a set of principles and process for handling variations and building adaptive architectures that are resilient to change. These extend and complement the notions and techniques of Enduring Business Themes [37], Adaptive Architecture [100] and variability and configurability [28][58].

2.1.2 Component Software

The literature reflects a lack of agreement and an overloading of component software terminology, offering a number of definitions of what a component is or should be. For instance, Clemens Szyperski defines a soft-ware component as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [25].

For example, a component provides prefabricated functional building blocks to be reused by rearranging them in new compositions. One example might be using prefabricated avionics software in a complex command and control application. In other words, components can be thought of as building blocks or units of independent deployment used for third-party composition and having no persistent state (there is no differentiation between the component and its copies). The majority of the definitions point out certain characteristics that are worth repeating.

First of all, terms in the literature that refer to a component (unit, piece of software, or abstraction) do not indicate any particular implementation technology. For instance, there is no need for a component to contain classes and be constructed using object technology, although that usually is the case. It could contain traditional procedures or it might be realized using any other approach and provide its functionality using any technology. Also, the term *unit* does not provide any indication about the size of the component. However, there are hierarchies of components, so size can vary considerably.

Second, the term *independent deployment* refers to the fact that components are typically unaware of the contexts in which they can be used. In this case, to be able to deploy a component independently means that a component needs to be well separated from its environment and other components. Therefore, a component encapsulates its constituent features, and it will never be deployed partially. This requirement usually has performance implications and is one problem when trying to employ such components in an embedded system. Third, if a component is to be used for composition then it has to be sufficiently self-contained with a clear specification of what it requires and provides. In other words, a component has an interface specification that describes what the component does and how it behaves when its functions or services are used. Through the specification, any potential user of those functions can use the component in his application without preoccupying

him-self with how those functions are actually per-formed. Also important is that a component can be viewed as a white box or black box building block depending on the visibility that the users have of its interface implementation. If a user has access to a component's source code, then it is said to be a white box component, since it implies some degree of extension and customization. If, on the other hand, a component is available with no source code, and may be used just as it is, it is described as a black box component. Finally, besides the specification of provided interfaces, components also are required to specify their resource and other needs. These needs are called *context dependencies*, referring to the context of composition and deployment required.

2.2 DECOMPOSITION

Parnas (1972) [72] pioneered the notion of hiding the information about design decisions behind module interfaces. The notion of program families is explored and introduced, again by Parnas in 1976. This was to be precursor for product-line architecture [51][25] and software kits. Parnas, Clements et al. (1984) added hierarchies and their documentation through Module Guides to render the criteria for module decomposition scalable.

Note that the main import of these papers was not as is commonly conceived: they were not solely about data abstraction – the hiding of data structures and exposure of methods to non-intrusively manipulate them – but rather about *encapsulating design decisions*. Variation-oriented Design [11] introduced by

Arsanjani in 2000 is built upon the same principles while marrying the above notions with the concept of design patterns introduce by Beck, Cunningham and Gamma [42].

This thesis extends these by a set of six principles, a process of externalization and a method by which VOAD can be achieved.

The literature contains few references on *how* to decompose a system or the actual mechanisms or methods for defining component boundaries; their identification and definition. The notion of decomposition around the axis of change or variability has deep historical roots. Parnas [72] presents the initial criteria for system decomposition around design decisions – later diluted into data abstraction and forerunning the notion of object-oriented software development. VanHilst and Notkin [131] extend this notion recursively to decomposition of modules themselves; rather than the systems. They note that minimizing the number of design decisions per module, although producing smaller-grained modules, makes the software more amenable to change. The authors do not explicitly consider the impact of design patterns [42], although we find a reference to Gamma et al. The module decomposition criteria they mention; i.e., of data representation changes, changes of behavior and algorithms, etc., can be elegantly handled by the notion of design patterns which are partially founded on the precept of variation-oriented design: "encapsulate what tends to change". Pree later

discusses hooks and hot-spots that can be used to provide a similar type of behavioral extension within object-oriented frameworks.

2.3 Reuse of Fine-grained Software Components

Fine-grained software components initially referred to as "*modules*" began to be evolved into abstract data types. Soon this evolved into the notion of *objects* and *classes*. With the realization that similar or virtually identical modules need not be rewritten for every project, the notion of reuse, first incorporated in program utilities and catalogs began to increase in importance. Classes were thus sought out for reuse. This was the first level of reuse. Johnson and Foote (1996) discuss the criteria to be used in designing reusable class's [77].

Classes soon began to be combined into groups of classes to form the second level of object composition: class libraries. The cohesion of these classes was defined by association, aggregation and inheritance or of the notion of delegation or composition.

This second level of class dependency was often used to form groups of coupled *object graphs* [132] around specialized domains such as graphics libraries, graphical user-interface widgets, financial instruments, etc.

Class libraries, being the second composition of classes, were utilities or advanced Application Program Interfaces (APIs) that a client program could invoke when a given functionality was desired. This type of class clustering led to a more specialized relationship between classes, which were based on inheritance and composition rather than merely a "*uses*" relationship.

Once the object graph's collaborations were generalized enough to represent key domain concepts as abstract classes and incorporate the collaboration of these abstract classes, often with initial default implementation, the resulting cluster was named an object-oriented *framework*. Subsequently, Roberts and Johnson (1997) present a *pattern language* for evolving frameworks [33].

The problem with frameworks lies in their strength: they abide by the principle of inversion of control whereby a client registers their software with the framework and the framework calls their program (classes that often implement or derive from a framework base class) at appropriate points.

2.4 COMPOSITION

The composition of the components of software architecture plays a more cardinal role in software engineering than the study of producing its individual components. Very often, the emphasis on research in this area is focused on the definition of component models such as CORBA

Component Model, DCOM, J2EE Enterprise Java beans, black box reuse; whether pre-existing (Custom-off-the-shelf (COTS)) or under development.

Thus the specification of this composition from both a structural (static), flow (dynamic) and adaptive (reflected and altered at run-time) perspective is highly important. This dissertation combines the well-known capabilities of domain-specific languages with the problems of component-based software engineering to achieve a highly re-configurable, adaptive architectural style.

This chapter further demonstrates how a domain-specific language is used to specify, generate and drive the configuration of a set of software components. A formal specification [119-121] is carried to component-base software architectures in the form of a formal language specification. The emphasis and usage context of this business domain-specific language is on highly re-configurable architectural style. Garlan et al., [43] describe the notion of defining architectural styles for a collection of related systems. The style determines a coherent vocabulary of system design elements and rules for their composition. By structuring the design space for a family of related systems a style can, in principle, drastically simplify the process of building a system, reduce costs of maintaining systems. In this sense, an architectural style is a meta-architecture; or architecture that describes the composition of a set of related architectures that arise within various

domains, but within the same context. Our last definition ties the notion of architectural styles to that of design patterns.[3]

The notion of using a domain-specific business language is introduced to define the static and dynamic aspects of the composition and configuration of component-based software architecture. The composition of a software system is more than the sum of its parts; its configuration is more crucial to meeting the objectives of reuse, maintainability and time-to-market than the sole focus on what components to buy or how to create the components themselves. Although current industry and academic focus has been on the selection of components, or of how to build them, little emphasis has been placed on how to assemble them into larger units of (e.g., business) functionality, especially while maintaining the ability to dynamically re-configure components at run-time.

Service-oriented architectures brought to the forefront with the advent of Web Services [94] provide the ability to dynamically bind to a remote component's services, invoke them and disengage on demand. This sort of run-time adaptation, of querying for components for services, binding to them or of creating a dynamic component assembly to realize a business transaction or session is becoming increasingly important. Current component technologies and concepts do not allow this.

---

[3] For a more detailed discussion refer to the section on Architectural Style and Software Architecture.

It grows increasingly more important to be able to realize an on-demand service using the dynamic composition of component-based software architecture with the goal of providing a set of well-defined services for a transactional purpose. E-procurement or satisfying a value chain of providers and consumers is an example use-case where for the purposes of a business-to-business supply-chain transaction or set of transactions, two parties dynamically locate and negotiate one another, with one or more parties assembling a set of capabilities or services required from them by a service consumer to fulfill a business transaction.

The duration of this composition or assembly may range from a few minutes to a semi-permanent one. If the given configuration tends to recur, it would satisfy non-functional or service level agreements to maintain this software composition over a period of time, where instances of components are assembled and collaborate in a configuration defined by a set of business objectives.

This paper draws from the vast literature and prior work in the fields of software engineering, domain specific languages, software architecture, domain engineering as well as the very well recognized and implemented theoretical foundations of compiler theory to construct a business complier. The steps to producing such a compiler call for augmentations to current

software development methodologies to support specification of business architecture and subsequently mapping it to composition-based component architecture.

Domain Analysis or Engineering is the process of defining, creating and evolving reusable assets for a family of applications in a domain. Often domain analysis [5] is used to identify commonality/variability as a first step in the direction of the definition of product line architectures (Clements and Bosch [24][25]). A formalism to depict its results is lacking such that could be used to generate, drive and execute a specification for a business domain under investigation [95].

Formal methods tend to have a steeper learning curve for the average developer or architect and add a significant amount of formal specification time to the life-cycle [119]. Such methods tend to be highly abstract and often cumbersome to implement in business and information technology (I/T) circles where there is a large degree of skepticism on the actual value-add that these methods and specifications bring to the end-product. The effort to integrate formal specifications into the life-cycle for business applications, render it so far removed from justifiable project activities that developers and architects (let alone project managers who are aiming to get the project some on time) shy away from producing them. Such, often impractical attempts are abandoned in favor of tried and true, yet potentially

more labor-intensive methods of software development using general-purpose programming languages (Java ™ and VisualBasic™ being currently in vogue).

This dissertation proposes a semi-formal method of software specification, development and maintenance through the representation of the manners of a system and its components in a domain specific language, resulting in the construction of a highly-re-configurable architectural. Thus, the notions of compiler theory (parsers, grammars) are combined with software engineering principles, software architecture, pattern driven software construction and domain-specific languages. Thus a balance is achieved by combining the strong points of more formal specification approaches with practical design and software architecture, pattern-related activities that directly produce workproducts of demonstrable value to the business sponsor of a software project. General purpose and special-purpose (domain specific) programming languages are thus used in a potent combination.

2.5 Component-based Software Engineering (CBSE)

Building software systems with reusable components brings many advantages. The development becomes more efficient, the reliability of the products is enhanced, and the maintenance requirement is significantly reduced. Designing, developing and maintaining components for reuse is, however, a

very complex process which places high requirements not only for the component functionality and flexibility, but also for the development organization.

This section discusses the different levels of component reuse, and certain aspects of component development, such as component generality and efficiency, compatibility problems, the demands on development environment, maintenance, etc. The evolution of requirements for products generates new requirements for components, if components are not enough general and mature. This dynamism determines the component life cycle where the component first reaches its stability and later degenerates in an asset that is difficult to use, difficult to adapt and maintain. When reaching this stage, the component becomes an obstacle for efficient reuse and should be replaced. Questions related to use of standard and de-facto standard components are addressed specifically.

Modularization was probably the first step in the direction of the design and implementation of software components. Even this is not an ad hoc process, the criteria for decomposition was often left to the experience of the system designer or programmer. However, the criteria against which systems are decomposed or partitioned into modules is critical to success. The hardware industry has many shining examples of effective component use as testified by the increasing success in producing replaceable parts that can be independently manufactured based on standards. Thus, in the

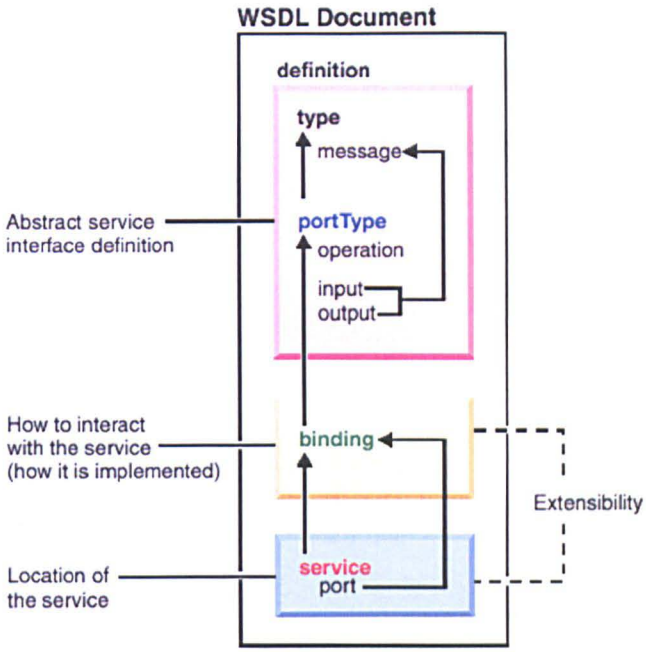
hardware discipline, component technology has been successfully in use for some time. This success has been a source of envy to software designers, raising question such as: "Why isn't software more like hardware?" Hardware-like *software integrated-circuits* [123], for example, have been suggested over a decade ago as a potential solution to the expected *software crisis* [140]. In fact, software engineering textbooks *(falsely)* cite hardware component reuse as working so well, that "component reuse is not regarded any more a central issue, since it is so obvious and widely practiced." But so far there has been little success applying this approach in reusing existing software. *The author addresses this problem by using a solution approach based on resolving issues in all five domains of component-based development and integration, and not merely focusing on technology implementation issues*; the traditional misplaced focal point of component-based software engineering.

A component is often thought of as an independent part or constituent element of a larger whole that is wired together through its connectors to its ports [30]. In most of the literature, the components themselves are first-class, not the context or configuration they reside within— contrary to the experiences gained in other disciplines. Although components should be replaceable they should not be context independent; they are bound by contracts that assume an underlying wiring mechanism which will facilitate a collaboration sequence to achieve the purpose of the component (and the services it provides).

This connotation, often structural, is also taken to mean a deployment-time binary unit or executable code that can be "loaded" and "run" or "linked" independently. The UML [84] defines components as deployment-time entities. Notwithstanding the work of the present author, *how a design gets mapped to deployment-time components is currently not a well defined process.*

In this dissertation a context-aware component (CAC) is a software entity exhibiting three main characteristics: presenting **services** (this can be web services, for example or merely methods on a class), abiding by **contracts** and exhibiting context-awareness or *manners* [14].

Meyer expounds on contracts and introduces and implements the notion within Eiffel [66]. Interfaces, taken form the world of object-orientation [22] and implemented in many programming languages such as Java, C++ and recently, C#, are a means to separate implementation from the specification of a method. More recently this ideal has been realized to an even larger extent through service-oriented architecture based on Web services. Web services provides a Web services description language (WSDL) [2] which provides a clean separation between the interface and implementation of a service.



Here, the extensibility of a service is through two degrees of freedom: namely, how a service is actually implemented and secondly on where the actual provider of the service resides (and thus the location of the service).

This is based on the following architectural style, namely the service-oriented architectural style. Note that a style of architecture has three essential elements: components, connectors and constraints. The components in client-server architecture, for example are the corresponding roles of the client and that of the server.

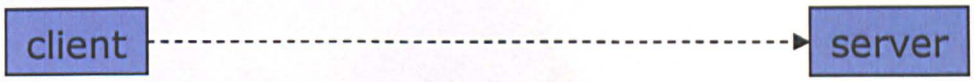


Figure 6 client server architecure

If we were to apply variation oriented design and externalize the ability to change servers without having to hardwire the connection between client and

server, we obtain a service-oriented architecture whose major components are the service provider, service consumer (or service requestor) and the service broker, who provides the level of indirection between the requestor and provider.

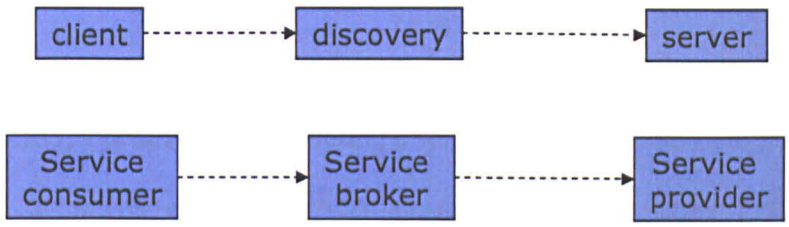


Figure 7: The Components of a Service Oriented Architecture

The second element defining an architectural style is the connectors between the components. In this case, we have several connectors that are shown as the methods or service types for each of the components as shown in the figure below.

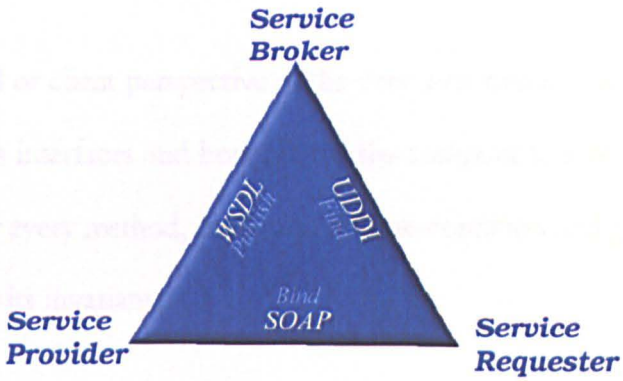


Figure 8: A service-oriented architecture

The EJB 2.0 specification introduces the notion of a message-driven bean (MDB) which marries the notion of message-oriented middleware (MOM) with Enterprise Java beans, solving the issues of asynchronous invocation and guaranteed delivery through applying the Java Messaging Service to EJBs.

Thus, components have evolved from being merely small-grained entities (e.g., a class, a Java Bean) to covering a wider spectrum: from being medium grained (a cluster of collaborating classes) and finally, larger-grained or coarse-grained units of business functionality, for example an account management, customer management or billing component [9][74][102].

2.5.1 Component Perspectives and Roles

The EJB specification classifies component producers, consumers into specialized roles [34]: the assembler, the EJB provider, the container provider, etc. There are two types of roles that lead to criteria for component specification: the external and internal perspectives.

The external or client perspective at the very least needs to know about the component's interfaces and hopefully, if the component is well-designed, its contract: for every method, determine the pre-condition and post-condition, and possibly its invariants.

D'Souza and Wills [30] utilize OCL (Object Constraint Language) [70] to introduce greater precision to the specification of object-oriented

frameworks, and to some extent to "components". They further generalize the notion of generic types or templates (a la C++) into the design domain by depicting how a framework is similar to the notion of the C++ template.

The Rational Unified Process provides stages and steps that tend to build smaller-grained components or objects. Coupled with UML, it uses the notion of subsystems to introduce an intermediary classifier between classes and packages. It does not directly support the notion of components starting from business modeling.

It is cardinal that components are designed and implemented in a consistent fashion within the same organization for maintenance reasons, not from a component's client's perspective.

Current methodologies do not provide first class support for component-based development and integration (CBDi) and component-based software engineering (CBSE) in general. Although the claim to components is there, the only notion that is directly supported as a first class construct is that of objects that have identity, state and behavior.

Some authors [2][136] have described the merits of interface-based computing and consider it to be the replacement for a component-based perspective. Supporting component based development and integration

(CBDi) across large enterprises requires support for each of the five domains of CBDi. Moreover, the emphasis on interfaces, although important, does not add more value than a type model for objects. D'Souza and Wills, Booch, Rumbaugh and Jacobson, Arsanjani, Gamma et al., emphasize the principle of separating interface from implementation.

However, the problem with the current definitions is that they are in the abstract: they provide no guidance for how to identify and scope component boundaries in a systematic fashion. This leads to a trickle down of irrational and unfounded component boundaries and object selection that leads to inflexibility in design. One of the contributions of this thesis is the definition of a method (activities and workproducts) full-life cycle support for component-based development and service-oriented architecture on a large scale. Also, the second related contribution is the definition of a set of patterns for component design. Now that the boundaries and semantics have been defined; we show how to design and build the component in a technology-neutral fashion. Components require an operational definition: i.e., how to build the component from a flexible design. The Enterprise Component pattern [9][18] provides such a focus for serving as a standard template for design across projects within large organizations.

Components expose and require services, are wired together, via their ports and connectors based on their contracts. The latter component capabilities are solely from an external component consumer's perspective.
A component service provider (CSP) must provide an abstract specification that although it exposes very little or none at all about its implementation provides a succinct semi-formal specification of the internal manners of a component.

Arsanjani introduces the notion of manners as the laws or rules governing the behavior of a component within a given context [14][16]. Objects have identity, state (attributes) and behavior (methods, services). Components on the other hand need to know how to behave differently depending on their context. Unlike a method or service invocation on an object, an Enterprise Component (EC) never responds blindly to a service request. Instead, it is an event-aware, message-enabled large-grained element.

The EC registers interest in an event (a publish/subscribe model) and thus introduces the notion of notification similar to Message-driven Beans in the Enterprise JavaBeans 2.0 Specification. When the event occurs, a broker such as a message broker may notify the EC that an event has occurred.

2.5.2 Component-based Programming

Issarny et al., (2000) [90] describe Component-based programming in the context of distributed systems. Szyperski introduces the notion of

component-based programming as being above and beyond regular object-oriented programming, requiring different notions of component instances, containers, interfaces, run-time characteristics [25].

Visual Basic started the trend for visual components or "widgets". The X-window system and motif contributed their fair share of disassociation of presentation and control epitomized in the Smalltalk Model-View-Control architecture and later, architectural pattern; two variations of which, Document-View (in Microsoft Document Models exemplified in MS-Office products) and Presentation-Abstraction-Control (PAC) [40] are popular and in wide usage.

But components are beyond presentation. They are more interesting on the server side, as exemplified by EJB and Microsoft .NET.

2.6 GRANULARITY

Although the terms used to refer to a component do not give any indication about its size, the right size is one that makes it most useful. This means a component must have some quality issues such as correctness, robust-ness, careful specification, and so forth. Also, a component must provide the right set of interfaces without restricting context dependencies. For example, a component should provide all required software encapsulated in it, but this would increase its size. On the other hand, a component could be designed to provide maximum reuse capabilities with a likely increase of context

dependencies. As both approaches present inconveniences, there has to be some balance in order to come up with the right size component.

First, component-based architectures are considered modular, and so naturally layered, leading to a natural distribution of functionality. This modular approach makes the dependencies more explicit, helping to reduce and control them. Therefore, modularity is a sort of precondition to defining components and their granularity. A system can readily be partitioned into units of varying size and coherence. Second, to achieve the best granularity of components, the rules governing the partitioning vary from case to case and may depend on many different aspects, such as abstraction, analysis, compilation, fault containment, and loading. Depending on these aspects, a component could have different granularity. For example, as a unit of abstraction, a component could be an abstract data type, such as a stack or a queue, while as a unit of fault containment and loading, a component could be an entire file system.

2.7 INTERFACES

A fundamental principle of component-based design is that a component has an inter-face. All connections between components occur through interfaces that can be defined as a set of functions invoked by other components. To guarantee component independence, component software maintains a strict separation between the interface specification and the interface implementation. The interface specification of a component is a well-known

contract specifying how a component's functionality is accessed. In addition, the specification provides the necessary information for both those implementing the interface and using the interface. Besides functional aspects, an interface specification may also contain non-functional requirements, such as performance.

To develop useful interfaces, understanding the behavior of the participants of key activities in a domain is effective. In this case, component modeling and domain modeling are helpful. A domain model sets the context for the area being studied, which can be a large area or a part of a specific application. The key thing about domain models is the possibility to point out and describe important components, their relationships, and the meaningful collaborations between them in the domain of interest. In component modeling, the interactions between components can be analyzed and captured, which is helpful for interface specification and its implementation. Interactions between components are called collaborations, which may be complex, involving many parties and an agreed sequence of actions between them.

The main elements of an interface are its list of functions with the corresponding parameters expected from its callers and the specification model that provides the means by which each service may be understood. How-ever, it might be necessary to have more information about a component to determine its behavior.s In this case, besides the basic con-tract,

which is composed of the functions, parameters, and possible exceptions, an inter-face through its specification model can contain another three levels of contract: behavioral, synchronization, and quality of service.

2.8 Tools and Infrastructure

To be useful, components must be implemented, assembled, and interact with other components. Therefore, they require tools that may be specialized to component assembly and construction, and they also require some basic support structure (infrastructure) providing the means for their interaction. First, it is helpful to know what kind of programming languages can be used in component software development and if there are some special requirements. For example, as component programming supports incremental loading of code, late binding has to be supported because interactions with other components need to be dynamic. Other features, such as polymorphism, information hiding, and safety also are meaningful. Languages such as C, C++, Modula-2, or Smalltalk are not truly component-oriented programming languages because they lack the support for encapsulation, polymorphism, type safety, module safety, or any combination of these.[7] However, almost any programming language can be used for developing components.

The development of component software appears to be more dependent on supporting tools. Although most of the traditional tools of software engineering for design, implementation, and maintenance will continue to be

used, new tools will be necessary. Today, most of the tools concentrate on component assembly normally performed by instantiating and connecting component instances and by customizing component resources. Some assembly tools assume that all component instances have a visual representation at assembly time and then use powerful graphical builder tools to assemble components. An important aspect in the assembly process is that it should be automated and repeatable wherever a modification is necessary regarding the availability of future versions of components. Finally, there needs to be some kind of environment that supports components conforming to certain standards and allows instances of these components to be attached into the component environment. This infra-structure should establish environmental conditions for component instances and regulate the interaction between component instances. All popular component infrastructures provide mechanisms that allow development in multiple languages and execution across multiple hardware platforms. Examples of such infrastructures include Corba (common object request broker architecture), COM (Component Object Model), DCOM (Distributed COM), and Sun's JavaBeans. As reusable components have been a trend in software engineering for some time, Corba, COM, DCOM, and JavaBeans all address these concerns. These systems serve important, but different needs than the ones addressed in this article. They provide a kind of macroscopic-level infrastructure for component-based software.

More recently, the Jini architecture has been defined to address the need to plug components worldwide into networks.[13] In Jini's world, the components to be plugged into a network can be large software components, entire applications, hardware devices, and embedded systems. The Jini system depends on and works with Java and consists of sets of interfaces. These interfaces include distributed events, a two-phase commit protocol, and various functions involved with resource allocation and reclamation. Also included is support to aid in supplying and finding services through lookup and discovery components. The system is very open-ended, as it needs to be to address worldwide networks and evolution.

A key ability in Jini is the dynamic plug in ability and concepts that support this capability, which may prove useful in some aspect of application specific operating systems where such kernels must support hot swappable soft-ware. A key difference between Jini and Corba, for example, is Jini's ability to download code to the client that is then used to communicate with the server. This approach permits changes to servers to be evolvable and be propagated to clients at the time they are to be used. Jini is also serving a different need than the one addressed in this article.

2.9 Relating Component Concepts to Object-Oriented Types

Much of the discussion of component composability, reusability, and substitutability can be linked to the terminology of object-oriented (OO)

types [21, 22, and 23]. Doing so serves a dual purpose: (1) it enables readers whose primary expertise is in the area of OO type theory to relate the concepts and terminology of CBSE to those with which they are familiar, and (2) it helps clarify the composition properties of components and connectors. For example, conditions specifying when one component may be substituted for another are akin to sub typing in OO programming languages (OOPL) according to the Liskov Substitution Principle (LSP). At the same time, the differences between the presented architectural concepts and typing in OOPL can help identify the limits of applicability of methods and techniques developed in one to the other. Specifically, objects tend to create object graphs that have relatively high coupling, whereas components tend to have crisper boundaries: they are more loosely coupled.

Conceptually, components and classes are similar but not identical. The services a component provides are equivalent to a class specification, and the requests it sends correspond to OO messages. However, no OOPL concept corresponds to event-based or message-based notifications, whereby state changes of enterprise components are reified as messages and no assumptions are made about the existence or the number of recipients of those messages. This results in the possibility of messages being ignored in enterprise component architecture, whereas a similar situation would result in a runtime error in an OOPL.

The distinction between notifications and requests and the topology the enterprise component style imposes on a set of components in architecture are the major differentiators between the component-based paradigm and OOPL. Nevertheless, the similarities between CBSE components and OO classes allow us to explore the ramifications of OO sub typing on reusability and substitutability of components. For that reason, we assume that a fine-grained component is a class in the OO sense, exporting two interfaces, medium- and large-grained components being Composites that aggregate other components or objects.

2.9.1 Issues in Enterprise Components

During the course of diverse client consulting projects across multiple industry contexts over many years, we have experienced five common threads of prevalent problems and issues.

Firstly, the overall *domain context view*, consisting of collaborating *subsystems*, is often lost in favor of tactical development needs such as database tables, user-interfaces and fine-grained objects. Successful wiring of components is conducted within a well-defined domain context view; describing how subsystems collaborate to ensure business goal fulfillment across business processes and workflow steps against business rules. This can often be done in terms of a *collaboration reification*, which defines a pluggable workflow that can be realized in terms of a Rule Object pattern [13], so the workflow steps can be adapted to new business requirements.

Secondly, in order to create a component, it is important to specify the services the component will *require* and *provide*. This *component interface specification* not only includes the externally visible behavior (services), but also the rules governing this behavior (component "*manners*") and meta-data needed for reflection and component *service discovery*. Thus, a holistic picture of an end-to-end business process model is necessary to show the subsystems and their perceived interactions within the business domain and software realization. This domain-partitioning and modeling of manners is the prime function of subsystem analysis, which is one of our extensions to current methods.

Thirdly, there is, most often, a *conceptual mismatch between the business and software models*. These models are created for different reasons by different teams. Usually, the business model is ignored altogether or "magically assumed" to exist [35]. Most object methods view a software modeling process as consisting of an identification of the identity, state and behavior of fine-grained classes in the problem domain based on ambiguous business requirements that are most often incorrectly "assumed" to exist [31]. Business modeling is mentioned in only the most recent versions of the methods [82]. This creates a curious impedance mismatch between the business model and the software architecture that will eventually realize a well-defined subset of the business model. The mismatch stems from a "conceptual gap" between an often vague and ambiguous under-specification of what is needed and what the development team assumes is meant, which in

turn, leads to unnecessary rework as tangible and executable extensible prototypes emerge with each iteration. The result is yet another problem of creating design elements that are too fine-grained, interdependent and not explicitly planned for reuse.

This leads to the fourth issue of *component granularity*: "from what level of granularity should we commence modeling and designing components?" It must be determined if one should start (object discovery vs. subsystem discovery) by defining whole business subsystems and processes (large-grained), parts of a business process (medium-grained) or more fine-grained business entities such as Customer, Account or Loan within them.

Finally, a *reuse level* is assumed or chosen as a starting point for modeling. A common misconception assumes that the fine-grained "class" is the ultimate unit of reuse and thus of "componentization." Our research suggests that higher levels of reuse (e.g., the "cluster" level) are more suitable for CBDI and creating resilient business-supportive software architectures.

2.10 Implications for Enterprise Application Integration and Workflow Systems

Many enterprise architectures are moving to a hub-and-spokes architectural style that utilizes message-oriented middleware to route and transform incoming data and messages among a set of applications. This Enterprise Application Integration approach [143] is useful as a first step in providing synergies between otherwise non-communicating, semi-autonomous systems. Current industry trends point to the need to provide the output of one business process as input into another one.

In many cases, this business flow, or the order in which information and messages are sent to various loosely connected applications or subsystems require the use of workflow management systems [137].

If manual intervention is necessary, workflow management systems such as MQ-Workflow ™ are sometimes employed to maintain state across invocations, provide context and guide the branching logic of where a document must be routed based on a number of business rules to the appropriate role in the company.

2.10.1 Micro-workflow and Macro-workflow

It is convenient and in most cases necessary to distinguish between three levels of workflow: macro-flow, medium-flow and micro-flow. The latter should not be confused with micro-workflows [139].

The macro-flow is the workflow or business flow between large grained elements such as subsystems or applications or enterprise components. Micro-flow deals with the collaboration of business logic among the internal representation of an enterprise component.

Thus, the axes of variation, determined by Variation-oriented Design[4] (VOD) [110] are encapsulated and externalized as axes of configuration [14]. A design pattern that defines this process is a Configurable Profile [9].

Each level has its own implementation mechanisms that allow the optimization of the architecture to conform to non-functional requirements and constraints that relate back to the patterns for symmetry and stability in software architecture [17].

To achieve stability amidst change, Van Hilst and Notkin (1996) [131] consider it a requirement to localize change: minimizing its impact across the system. This decreases the effect of architectural entropy that increases with every unanticipated change made to the application (logical) and technical (physical) architecture. This is an extension of Parnas' [72] proposed information hiding principle to encapsulate changing aspects of a module and hide the design decision from outside the module.

[4] See section on variation-oriented design.

Thus in this thesis, the contribution is the method steps (VOAD, Externalization) that takes into account the variations arising in the domain and including them as part of Business Analysis and Architectural Analysis.

Ultimately a system consisting of components must map it's logical component model to a physical operational realization; or an operational model [47][48]. In this model components are allocated to nodes in the physical architecture and the architectural mechanisms become relevant and must be applied [48].

This problem is solved in GOOD by the introduction of manners that encapsulates the functional and non-functional aspects of the components (micro-level) and their interactions (medium-level) and collaborations (macro-level). The process of introducing manners is outlined in the method proposed by GOOD. Also, the architectural style and techniques needed to support it are outlined.

2.11 Review of Domain-specific Languages

A *domain-specific language* (DSL) is a programming language or executable specification language tailored specifically to an application domain: rather than being general purpose it captures precisely the domain's semantics and offers the notations and abstractions necessary to express the semantics of

particular domain [1, 4-7,76]. Bentley describes them as *little languages* [53] and gives examples using PIC and other languages used to built it (*lex* and *yacc*) and discusses the DSL design method. A DSL-based development methodology addresses the need for increasing domain specialization in the software engineering field [1]. Domain-specific languages allow the concise description of an application's logic reducing the semantic gap between the problem and the solution program [4].

DSLs have been built for literally hundreds of domains [3]. Examples of DSLs are outlined below. In particular, more familiar ones include *lex* and *yacc* used for program lexical analysis and parsing, HTML used for document mark-up, SQL the structured query language, BNF (Backus Naur Form) used to describe grammars for programming languages and VHDL used for electronic hardware descriptions. The key characteristic of DSLs according to this definition is their *focused* expressive power [5][7].

The domains themselves can be grouped into the following areas [1]:

- Software Applications
- Systems Software
- Graphics and Hypermedia
- Telecommunications]

- Artificial Intelligence
- Mathematics
- Hardware Design

Moreover, Deursen et al [1] contrast a "domain as the real world" point of view as adopted in the artificial intelligence community, with a "domain as a set of systems" approach, as used in the systematic software reuse research community.

DSLs are usually *small*, offering only a restricted suite of notations and abstractions. In the literature they are also called *micro-languages* and *little languages*. Sometimes, however, they contain an entire general-purpose language (GPL) as a sublanguage, thus offering domain-specific expressive power *in addition to* the expressive power of the GPL. This situation occurs when DSLs are implemented as *embedded languages*. Languages such as Cobol or Fortran, which could be viewed as languages tailored towards the domain of business and scientific programming, respectively, are generally not regarded as DSLs, because they are not small and because their expressive power is not restricted to these domains.

Domain-specific languages are usually *declarative*. Consequently, they can be viewed as specification languages, as well as programming languages. Many DSLs are supported by a DSL compiler which generates applications from

DSL programs. In this case, the DSL compiler is referred to as *application generator* in the literature [3], and the DSL as *application-specific language*. Other DSLs, such as YACC or ASDL, are not aimed at programming (specifying) complete applications, but rather at generating libraries or components. Also, DSLs exist for which execution consists in generating documents (TEX), or pictures (PIC). A common term for DSLs geared towards building business data processing systems is 4th Generation Language (4GL).

Related to domain-specific programming is end-user programming, which happens when end-users perform simple programming tasks using a macro or scripting language. A typical example is spreadsheet programming using the Excel macro-language.

The contribution in the area of DSL's is to employ the use of DSL's, not merely in the traditional sense of usage, but extending it to include a two-level specification and run-time combination of domain-specific and general – purpose languages used in different steps of the method used to create the software system, such that the general-purpose code is used in the detailed computations that are best achieved by general purpose programming languages, and the higher order semantics of composition, flow and navigation are expressed declaratively by the domain expert who is more knowledgeable than the technologist in the business domain, as a domain-specific language; e.g., a context-free grammar that drives and invokes the

decoupled services of the components or general functionality available to perform the business logic, presentation and manipulation of data.
Although the languages are separate, they are defined to be related and loosely coupled making modifications simpler and less time consuming.

Secondly, another contribution in this area is that since the business grammar that implements the manners is a runnable specification that drives the invocation of the computational specific code written in the GPL (general purpose language), the need for code re-engineering is minimized.

2.12 Conclusion

The problems and issues described in this chapter are seen to be present in the domains of CBDi (see Table 1) and cover areas that form a spectrum from concrete practical tools to high level strategic business models.

They all are deficient in one common notion - that of dynamic re-configurability and its implications in terms of externalization, manners, context-aware components, methods and models.

The notion of integration of data, process and tools in terms of Enterprise Application Integration, data transport between tools, etc., although a necessary condition, is no longer adequate to take an enterprise to higher levels of integration enabling on-demand integration within a value-net.
In the subsequent chapter we offer the contributions of Grammar-oriented Object Design as a solution. We then elaborate on this solution in Chapters

Four and Five and describe manners and executable specifications along with an implementation of a tool supporting the concepts of GOOD.

3 CHAPTER THREE: GRAMMAR-ORIENTED OBJECT DESIGN

Objectives

- To define the process of Grammar-oriented Object Design (GOOD)

- To describe the concepts underlying GOOD and how it is the junction point for eight disciplines of software engineering

- To introduce the notion of externalization, variation-oriented analysis and design and how change can be accommodated into a software architecture

3.1 GRAMMAR-ORIENTED OBJECT DESIGN (GOOD)

GOOD is a new discipline of software engineering that combines and enhances and introduces innovations at the junction point of several software engineering disciplines. It lends a holistic and unifying perspective to a set of loosely related disciplines. GOOD includes a methodology (process model and method) to produce component-based, service-oriented and dynamically re-configurable architectures. These three styles of architecture form a spectrum and as we move from components to services to re-configurability, based on the needs of the domain at hand and the project applied to, an increasing degree of flexibility, business driven maintainability, reuse and re-configurability is achieved. These attributes are conductive to the realization of the on-demand paradigm of computing.

The method and reference architecture (DyRec) alleviates the problems outlined in Chapter One. Within the methodology (outlined in Chapter Ten) the notions of externalization [10] and variation-oriented design are used to augment the definition of manners for a dynamically re-configurable architectural style.

3.1.1 Usage Scenarios

GOOD can be applied to three types of scenarios:

1. Static aspect. How to compose a set of components; component or service assembly is declaratively defined by a domain-specific grammar.

2. Navigational aspect. The interaction between tiers in an architecture often requires the traversal of remote object graphs. A grammar defines the traversal path of the objects that are relevant versus the entire exhaustive search that may be otherwise conducted.

3. Dynamic aspect. The flow composition or choreography of the collaboration between components is documented as a business domain-specific language and externalized in the configurable profile that

3.1.2 GOOD as the Junction point

GOOD can be seen as the unifying element present at the junction point of various disciplines in software Engineering concepts in various (see Figure 9 and Figure 10). It impacts not only software architecture and the applications that run on that architecture, but empowers the business analyst, modeler or architect to play a more significant and important role in the software development lifecycle. Software applications are built on software architectures. Both should be created in order to support business function within commercial or scientific enterprises. Thus, the modeler's role is elevated from the requirements writer to that of the business architect who is empowered to create an executable representation of their business processes that can be mapped and still plays a significant role in the software development process.

GOOD affects and is affected by other disciplines within computer science, specifically, within the realm of software engineering. These are shown in Figure 9: Related Disciplines and their gaps filled by GOOD in the following page.

To bring cohesion to these multiple concerns and areas, an integrated methodology is therefore needed, that threads through these disciplines and brings wholeness. Figure 9: Related Disciplines and their gaps filled by GOOD describe the relation between various disciplines and grammar-oriented object design.

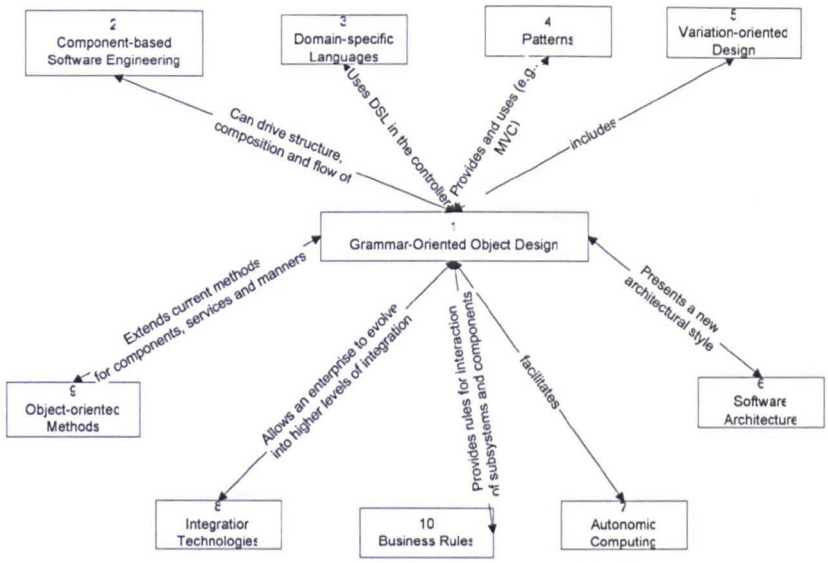


Figure 9: Related Disciplines and their gaps filled by GOOD

Table 1: How GOOD relates to other disciplines in Software Engineering

1. GOOD	Map business architecture to component-based software architecture using a domain-specific language approach.
2. Component-based Software Engineering	GOOD can provide an executable specification for the structure, composition and flow of components
3. Domain-specific Languages	Uses DSL in the controller
4. Patterns	GOOD provides and makes use of patterns at various levels of analysis, architecture and design
5. Variation-oriented Design	Extends current methods for components, services and manners
6. Software Architecture	Allows an enterprise to evolve into higher levels of integration
7. Autonomic Computing	Context-aware Components defined by GOOD Facilitate the design and creation of autonomic software components that fully support autonomic computing, not just from a hardware perspective but from an software application perspective.

8. Integration Technologies	Provides a roadmap for an enterprise to evolve to higher levels of integration
9.Object-oriented Methods	Presents a new architectural style
10.Business Rules	Provides rules for interaction of subsystems and components

The junction of the disciplines are seen in the following figure:

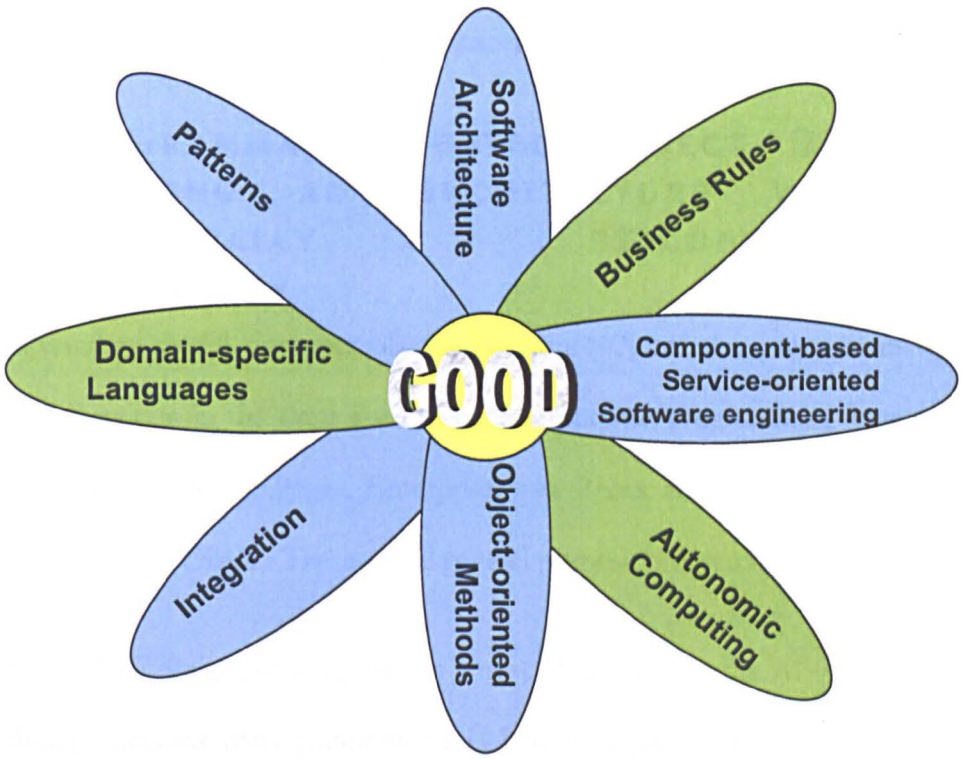


Figure 10: GOOD is the junction of several software engineering disciplines

GOOD provides the underlying wholeness in connecting apparently unrelated parts of these various disciplines in software engineering, together. It unifies object-oriented methods by introducing the extensions necessary for the development of component-based (from a service provider perspective) and service-oriented architectures (from a service consumer perspective) and enables the creation of a unique dynamically reconfigurable software architecture that abstracts and externalizes business rules and policies into manners that can be implemented by, patterns on one end of the spectrum and domain-specific languages at the other end of the spectrum.

The following section shows how the business grammar created by the business architect using their own (industry specific) domain-specific language will be used to drive the software architecture and applications that run on it.

3.2 Grammar-Oriented Object Design: Creating an Architecture with a Dynamically Re-configurable Controller

A standard model-view-controller architecture [40] includes components that are all written in the same level of general purpose programming languages such as Java Server Pages, Enterprise Java Beans and perhaps a backend system with COBOL. These are all general purpose programming languages.

"The MVC design pattern provides a host of design benefits. MVC separates design concerns (data persistence and behavior, presentation, and control), decreasing code duplication, centralizing control, and making the application more easily modifiable. MVC also helps developers with different skill sets to focus on their core skills and collaborate through clearly defined interfaces. An MVC design can centralize control of such application facilities as security, logging, and screen flow. New data sources are easy to add to an MVC application by creating code that adapts the new data source to the view API. Similarly, new client types are easy to add by adapting the new client type to operate as an MVC view. MVC clearly defines the responsibilities of participating classes, making bugs easier to track down and eliminate [141]."

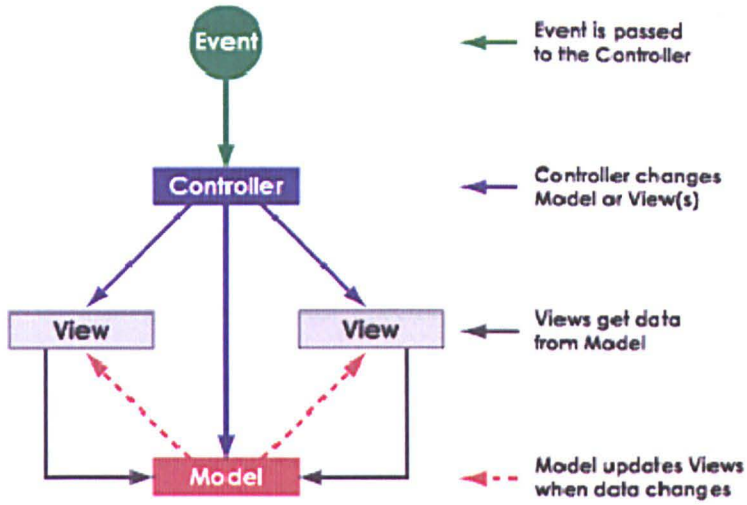


Figure 11: Standard Model-View-Controller [126]

The typical interaction sequence is shown in Figure 12.

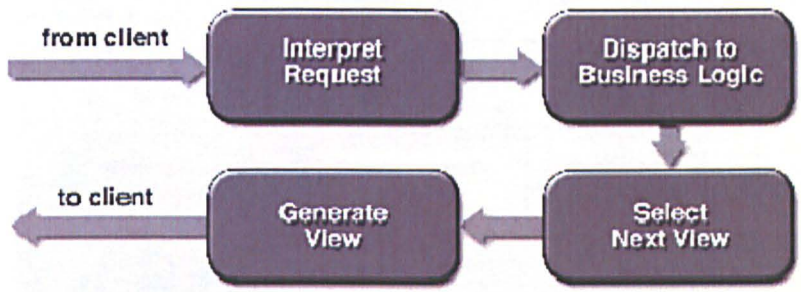


Figure 12: MVC Interaction Sequence [141]

This is a special case of a 3-tier architecture which is in turn a special case of n-tier architecture [142].

A GOOD architecture (dynamically re-configurable) on the other hand is unique in that it combines two levels of languages: the controller written a domain-specific language written by the domain expert and the view and

model written in general purpose programming languages. The controller layer (see Figure 13: Dynamic Controller Architecture Using GOOD) contains a dynamic interpreter that processes Business Grammars to transfer control to the appropriate component on an as-need basis. Typically this controller is written in a domain specific language versus the traditional notion of a controller which uses a general-purpose programming language implementation for the controller.

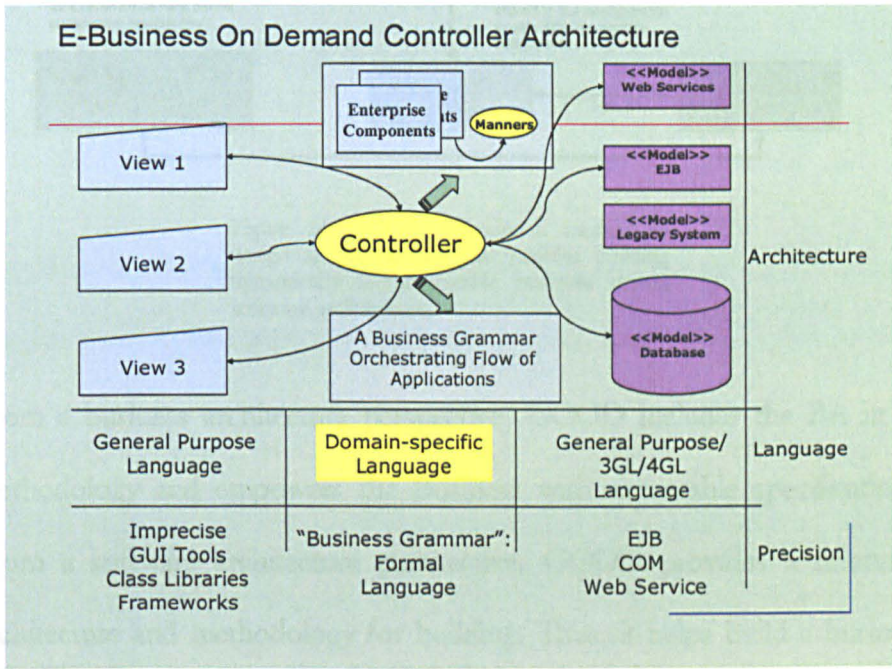


Figure 13: Dynamic Controller Architecture Using GOOD

Further, a conceptual model of how GOOD impacts business and software architecture can be seen in the following diagram.

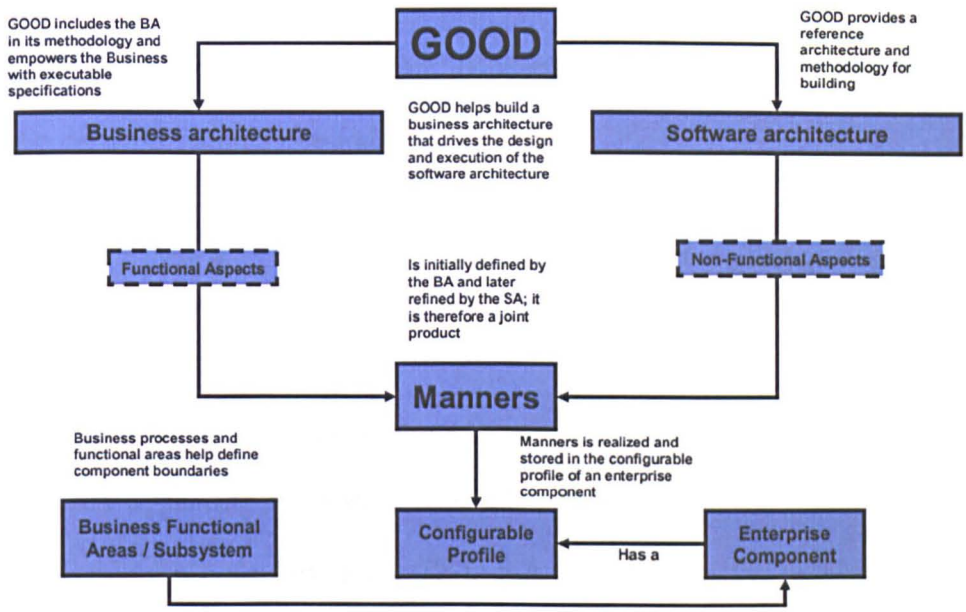


Figure 14: GOOD provides a method to design an architecture that enables building dynamically re-configurable business driven software architectures

From a business architecture perspective, GOOD includes the BA in its methodology and empowers the Business with executable specifications. From a software architecture perspective, GOOD provides a reference architecture and methodology for building. Thus, it helps build a business architecture that drives the design and execution of the software architecture. Thus the business architecture primarily defines the functional aspects while the software architecture also defines the operational or non-functional aspects of performance, availability, security, etc.

Manners are initially defined by the BA and later refined by the SA; it is therefore a joint product. Manners is realized and stored in the configurable profile of an enterprise component. Business processes and functional areas help define component boundaries.

3.2.1 The Specification and code ride together

Typically, specifications are analysis and design-time activities that ultimately disappear and are untraceable from the code. In GOOD, this problem is alleviated by having the specification of a program and its code "travel" together in the code (two levels DSL and GPL) and as the program runs (deployment time). The domain-specific languages used in its flow composition (between larger components) and inside its enterprise components will be driven by business modelers, with occasional help from architects.

IT staff will create general purpose programming language code for view and models (e.g., in Java) and the executable program will have both levels of languages within itself: the domain's language as well as the technology language.

Properties formally defined by the grammar will be ensured to be true in the running code because it is the domain-specific language that is running and orchestrating the flow and composition of components in this dynamically re-configurable service oriented architecture.

4 CHAPTER FOUR: MANNERS

Objectives

- Describe the need for representation of abstract collaboration semantics
- Describe the notion of manners
- Describe the notion of context-aware components

4.1 THE EVOLUTION OF MANNERS: REPRESENTING SEMANTICS OF COMPONENT USAGE

The composition, flow and usage of components and services are not explicitly configured in applications. This presents a problem with maintenance. *Manners* solves this problem by combining the information/specification necessary to use, compose and interact with the component in an externalized specification. Manners combines three of the best-practices outlined below to provide an executable specification in a domain-specific language.

4.1.1 Vertical Partitioning Through Encapsulation of Design Decisions

As we follow the evolution of the notion of software components, the encapsulation of design decisions is indeed one of the first and foremost criteria for componentization. Parnas describes this in his seminal work on the criteria to be used in partitioning systems into components [72]. Despite common misconception, Parnas does not exclusively introduce data abstraction; the example in [72] is an example of a broader principle, namely, the *"encapsulation of design decisions."* Let us name this type of partitioning, a vertical partitioning simply because the criteria are often focused within the context of a single project.

A second important best-practice is that of "separation of concerns". *Separation of concerns* is a fundamental concept of software engineering. It refers

to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern (concept, goal, purpose, etc.). Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Thus, it is the primary motivation behind componentization. This is often an additional consideration to the encapsulation of design decisions and constitutes a more horizontal or "cross-cutting" aspect of a software system. This quickly leads to a generalization in terms of multiple dimensions of separation of concerns (MDSOC) [98]. This allows us to refactor based on a set of often rapidly changing concerns: functional, non-functional, etc.

Once a system has been decomposed using these techniques (e.g., based on design decisions and separation of concerns) we now need to access these components. However, explicit access via method calls is not necessarily the most conductive to extension or maintenance. The context of the invocation or problem itself may change and the constituents that we implicitly rely on may no longer be there in "the software flesh". Rather, new components who still abide by the same protocols and implement the same interfaces (i.e., play the same *roles*) may still be allowed to participate in the collaborations defined within the system to fulfill its objectives (e.g., business goals within a business process).

Therefore, a third best-practice of software engineering is to "program to interfaces rather than implementations". This precept is expounded in detail in the literature of software design patterns [42].

4.1.2 Connection of Manners and GOOD

GOOD is implemented as system that is comprised of a set of large-grained **context-aware** enterprise components and services whose **manners** are declaratively described using a domain-specific language.

It must be noted that Manners can be implemented in a spectrum of realizations, each with increasing flexibility and re-configurability: as patterns such as Strategy, combination of patterns (such as mediator, state, strategy, etc.), a regular expression, and finally, the way we propose to use it in GOOD, as a domain-specific language.

Changes to the components, and the services provided by the components are made for the most part through re-configuration of their Configurable Profiles. Therefore, changes or enhancements to functional or non-functional aspects of the application or system is done non-intrusively thereby retaining the symmetry and stability of the system architecture while supporting the new business goals, objectives and needs.

Manners can be implemented in a spectrum of realizations: from the application of design patterns such as Strategy all the way to the realization as a domain-specific language describing the composition and flow of the

software component internally or describing the external collaborations of the components.

4.1.3 Impact of Manners on Software Architecture

GOOD advocates the use of an abstract specification of the semantics of a software system and the semantics of the composition of its components in the form of manners. Manners can be implemented using a spectrum of implementation decisions and realizations, starting from a Strategy pattern which encapsulates a family of algorithms and makes them interchangeable, to the externalization of process composition in a domain-specific language.

But one of the impacts of the separation of concerns of the composition and collaboration of (software and /or hardware) components is the architecture of systems built according to GOOD. Figure 15 below shows the relationship between a Model-View-Controller architecture and its variation as a GOOD Controller Architecture. Here, the manners of the system is implemented by the Controller using a domain-specific business language. The View and Model continue to be written in general purpose, (often) third-generation programming languages such as Java. Thus, M and V are in a general purpose and C is implemented in a domain-specific language.

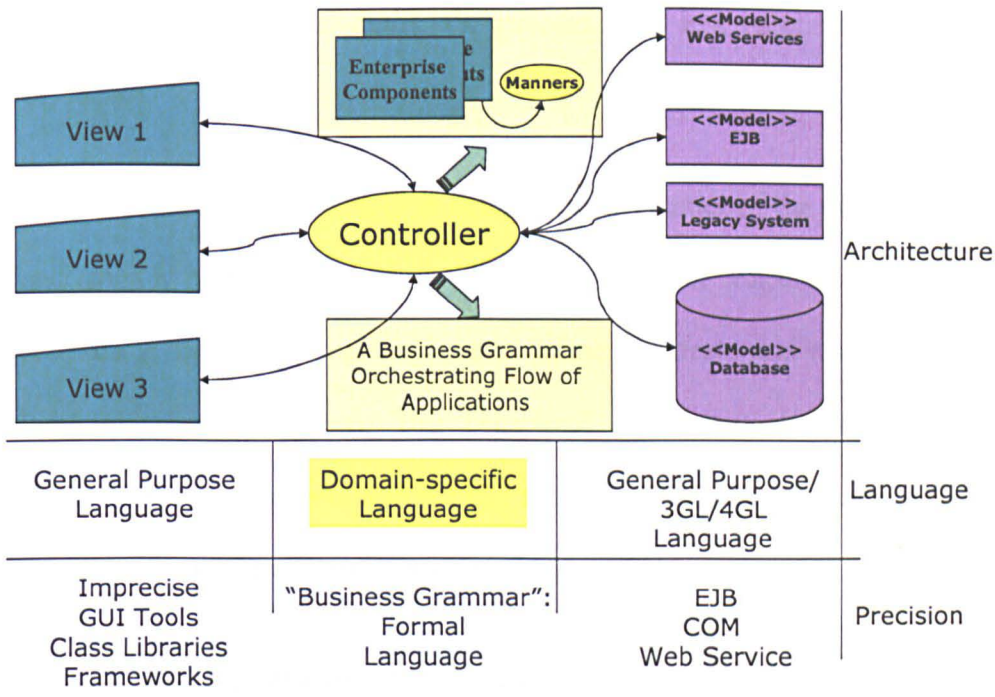


Figure 15: GOOD Controller Architecture

4.2 MANNERS

One of the characteristics of a software component is that a component is a deployable unit of functionality that can be assembled with other components to construct a system or application. More importantly, a component exposes its interfaces to a set of internal objects that it "bundles" and often hides (as in information hiding a la Parnas). Meyer [66] introduced the notion of contracts using pre-conditions, post-conditions and invariants.

Contracts specify the conditions that must be true before the invocation of a method on a component and what must hold true after the invocation has finished its processing, as well as those invariant aspects or properties that

should remain unchanged during the process. This is a component usage perspective.

However, contracts do not help in defining the semantics of the component itself; namely how the component is going to fulfill or realize its contracts (note that a component may implement various interfaces and may have to abide by several contracts). This behavioral specification is an abstract specification not a concrete one; it must allow the realization of the contract without constraining it to an implementation.

Thus, this challenge in component engineering is the difficulty in defining a highly adaptive abstract specification for the internals of a component from a service provider's perspective; the service consumer assumes the contracts and the provider must know how to build the contracts in a fashion that is conductive to maintenance. Namely, the externalization of functional and non-functional aspects of the component through variation-oriented design allows another dimension to the notion of separation of concerns.

Although the very nature of components point to the separation of interface from implementation, it is important to maintenance that the composition and behavior of a component under different contexts is amenable to extension without extensive internal modifications. Meyer calls this the "open-closed" principle [66].

In order to highlight the necessary characteristics that a component must support, it is convenient to contrast the behavior and characteristics of a component with that of a standard class or object in the object-oriented paradigm. When an object receives a message, it often responds to it by executing an action indicated by the message signature: the method is needed by a client program (object) and it is invoked.

Large-grained components, or Enterprise Components [6][13], however tend to be slightly more context-aware or exhibit more intelligent behavior: they do not blindly respond to a message by executing an action. Instead, upon receipt of a message or other trigger, it will first check its context, state and then decide which subset of rules to apply and after checking the filtered subset of rules for the validity of certain conditions, it may then execute the requested action.

To implement a dynamically reconfigurable architecture as proposed by GOOD, components and services need to be "context-aware". To do so, they need to be policy-driven; rule sets need to be selected based on the context in which the component finds itself. Components and services need to be aware of the distinction between valid and invalid combinations. These combinations include (typically a declarative mechanism of expressing the following; often implemented as a grammar):

- **Compositions**

    - " [As an application or larger grained component] How am I assembled?"; "What assemblies and compositions can I participate in?"
- **Collaborations**
    - "How do I engage in interactions and transactions with partner X?"; "Who can I partner with?"; "What are the rules of engagement?"
- **Contextual Behavior**
    - "Am I in a secure, transactional context to enact a transaction (e.g., submit a transfer from one account to another)?"

Thus context-aware component services drive dynamically configurable architectures as defined by GOOD.

When a component or service exhibits such context-sensitive behavior it is called a context-aware component or service. Context-aware components exhibit manners. Manners are the way a component behaves within a given context.

Manners consist of six main elements:

- **Context** – the domain related types that will determine policies, such as customer type (e.g., platinum, gold, normal), buyer type (wholesale, retail), etc.

- **State** – The current values of variables that are often passed in to be checked by a condition of a business rule.
- **Policies** – The meta-rules that determine which set of rules to apply within a given context. This serves as categorization and determination mechanism for sets of related rules.
- **Rules** – the event/condition/action trio that governs the business logic in applications.
- **Meta-data** (generally in a Configurable profile) – can be static or dynamic. The static aspects define the way a larger-grained component should be composed, the dynamic aspect shows how a set of components or services should be orchestrated via a domain-specific language using GOOD. Meta-data can contain functional and extra-functional aspects: how a component should behave and the quality-of-service criteria (non-functional requirements) it should conform to.
- **Actions** – the results of applying a business rule and finding a condition valid often entails performance of an action or the initiation of a workflow.

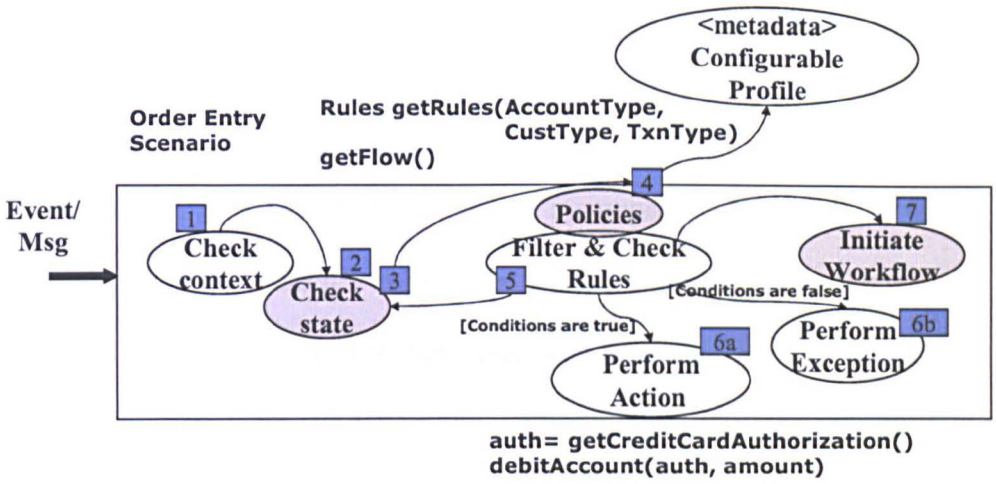


Figure 16: Manners defines Context -aware Component Behavior

XML can be used to represent the Configurable profile and its static, dynamic aspects. Also both business and technology aspects can be captured in a Configurable Profile.

As an example of combining meta-data with application components, EJB deployment descriptors contain information about the binding of an EJB to its environment.

The Configurable Profile pattern (Part of the CBDI Pattern language) defines a profile for each user type, so you can change the rules governing how the user type interacts with the system, their workflow, the rules that they have to abide by when they are offering products and services in a given office in a given geographic region (context) for a corporate (customer type) client within these time frames (duration constraint).

Therefore, a component is context sensitive: it will behave appropriately in each different context.

In this fashion, we introduce the notion of a component context to denote the background into which a component will be placed and is expected to function. The context has both functional and non-functional aspects. The functional aspects comprise the set of services satisfying feature requirements while the non-functional aspects of the requirements ensure usability and continued value to the users through security, performance, scalability, interoperability, maintainability, flexibility, etc., as defined in the non-functional requirements document.

4.2.1 Manners of Self-Aware and Context-Aware Components

4.2.1.1 Definitions

Context-aware components are components that have externalized their manners and use it to determine their behavior (e.g., in response to an event) based on the context they find themselves within. Self-aware components add learning ability to context-aware components. Thus, the self-aware component learns from the results of acting in specific behavioral modes when it responded to a given context. It can then use this to modify its behavior in similar future contexts. It achieves this through the application of artificial intelligence techniques for learning such as neural networks or learning engines.

4.2.2 Solution of Problems through Manners

A component in an enterprise context should be context-sensitive and pick-up the domain-specific behavior in context when it is operating within the confines of a given component context. Therefore, manners are a form of externalization of context-sensitive behavior from a component.

This allows non-intrusive changes to be applied to the component's flow, rules and behavior without having to modify the internal structure of the component; just its manners need to be changed.

Another, second, issue that manners solve is that of how to use, or defining the semantics of utilizing a classical interface in a programming language providing abstract data types. For example, given a set of method signatures, is there a semantic dependency inherent within the permissible sequence, alternative and repetition of invocation of operations, meaningful to that domain. What is the flow that semantically ties together the valid ways in which a set of API's (interfaces of multiple classes) or interfaces of a given class are to be used? Manners provide that answer in the form of a formal specification of their flow and usage.

In addition, certain pre-conditions have to be satisfied prior to invocation of a method and another set are guaranteed to be valid after its invocation– similar to the notion of software contracts.

Although contracts provide the valid state before, invariants during and permissible state after the invocation of a method, manners provides the abstract specification for the realization of the contract as a whole; not from an implementation perspective but from an abstract one. The actual implementation of the manners can be conducted in a variety of ways: hard-coded all the way to using a grammar to represent the manners.

Interface is separated from implementation so multiple implementations may be created to fulfill the interface and a hitherto unachieved flexibility if thus gained. Similarly, it can be said that manners provides a semantic separation of concerns that is sought when we separate interface from implementation. Interfaces however provide little or no semantic constraints on the way a class or type should behave as a whole (not just within the context of the contract of a single method)

A third area is that of **context-aware components**. Object technology is becoming the technology of choice for programming ubiquitous computing devices and their supporting environments. In pervasive computing domain, context is an important and powerful concept, referring to the physical and social situation in which objects are embedded.

A new set of issues must be confronted in building context-aware applications in new and emerging domains such as pervasive computing other than merely the business systems domain. The notion of component manners is a key solution to many of the issues inherent in developing such applications and

their supporting environments. For example, components in a pervasive computing system context as well as business systems needs to represent and interpret contextual information.

4.2.3 Levels of Integration

Enabling levels of integration to achieve on demand e-business in a utility-based model of computing requires the adoption of infrastructure necessary to support dynamic re-configurability. This dynamic reconfigurability increases with each level of integration outlined in Figure 17.

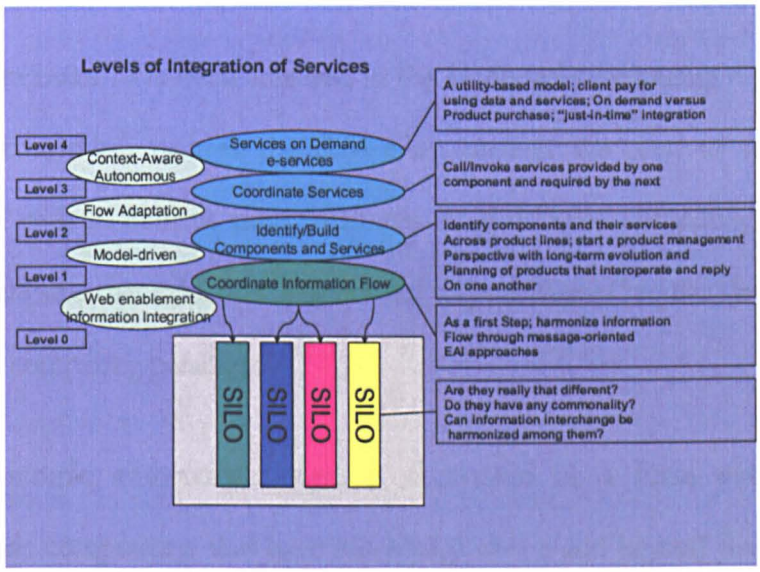


Figure 17: Levels of Integration of Services

[18]

4.3 CONTEXT-AWARE COMPONENTS

Computing systems that can configure, tune and repair themselves are called "autonomic." Let's dissect these attributes and explore their significance with respect to software application and architecture.

The three main notions are:

- Self-Configuration
- Auto-tuning
- Auto-repair

These attributes have been achieved in the autonomic computing domain of hardware systems, such as the IBM Eliza project. We need to explore a practical path to achieve these attributes in a software computing system, specifically one that exhibits the agility and responsiveness required by the on-demand computing paradigm.

An autonomic computing system is composed of a finite number of autonomic components that have knowledge above and beyond the average object: they know how to collaborate with one another and how to form valid compositions and constructs based on rapidly changing and on-demand specifications.

Thus, the components are context-aware (CAC): they can distinguish between valid and invalid combinations and collaborations. In order to know their colleagues within a collaboration, the CAC's need to know:

Who they are interacting with; requiring the other CACs to publish their service interface, contracts and manners

The context in which they are being "dropped" as if they were "dragged and dropped" onto a domain canvas will determine their rules and behavior. The context and state will limit and redefine their the boundaries in which policies must be applied and rule sets must be chosen for applicability and subsequent action taken.

4.4 THE GAP

The world of e-business on demand covers a wide spectrum. On the one hand are resource usage as utilities to maximize computing power, storage and infrastructure and on the other end of the spectrum are the capitalization of application level services that can be defined, re-configured, assembled and delivered on demand with "just-in-time" integration capabilities.

The promise of Web Services as an enabling technology that will enhance business value through providing capabilities such as e-services on demand will transform the way business is conducted and products and services are

offered over the web in communities of interest: business partners, customers and even competitors (e.g., auctions).

There are two perspectives relating to Web services as depicted in Figure 18: Two Perspectives on Web services . The "outside-inwards" perspective is the user's perception; no components, only services available. Once on-demand e-business is realized at the application and business level, service availability will be on-demand rather than static and pre-determined.

In the "inside-outwards" perspective the I/T department that provides the service must do so based on some form of container, for management of the services and correct partitioning and alignment with the business.
The enterprise component providing the services becomes an Enterprise Component Services (ECS) provider.

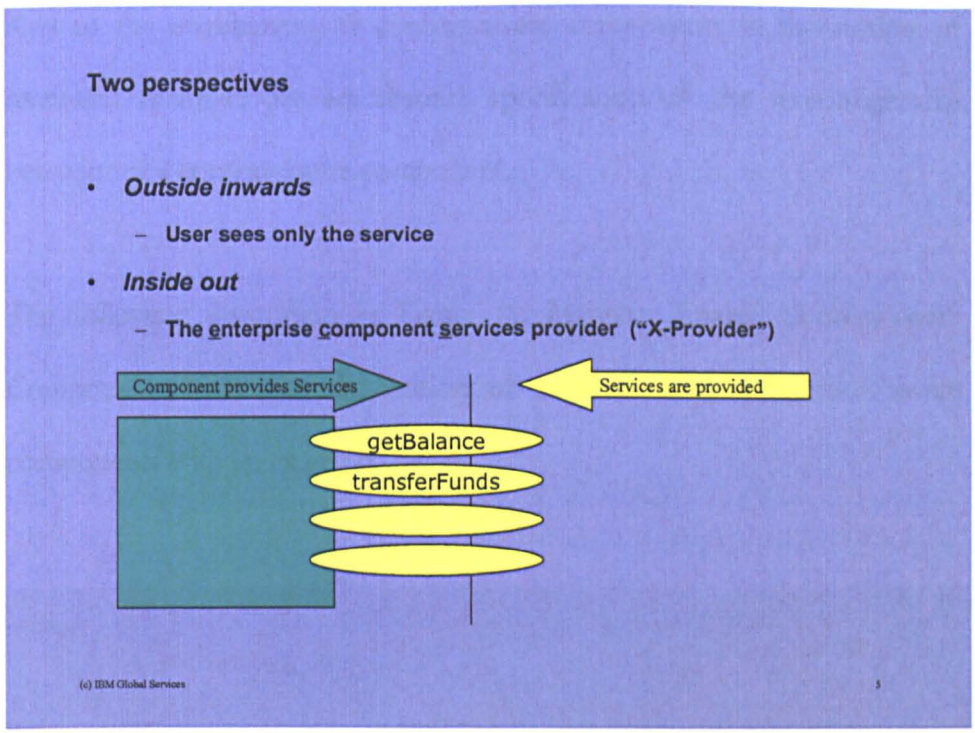


Figure 18: Two Perspectives on Web services

However, the world of on demand e-business and the realm of Web services need a very explicit link in order to enable business value derived from application level versus infrastructure level on demand e-business.

This gap is filled with the notion of context-aware components; or components that have manners. A context-aware component (CAC) can be rapidly assembled with other CAC's to provide just-in-time integration of applications to deliver new competitive value, not delivered with the previous configuration based on a previous set of requirements.

Key to the enablement of context-aware components is the notion of manners. Manners are an abstract specification of the re-configurable behavior of a context-aware component.

The following illustration in Figure 19: Manners Enable Context-aware Components, describes the notion of manners enabling context-aware components with an example.

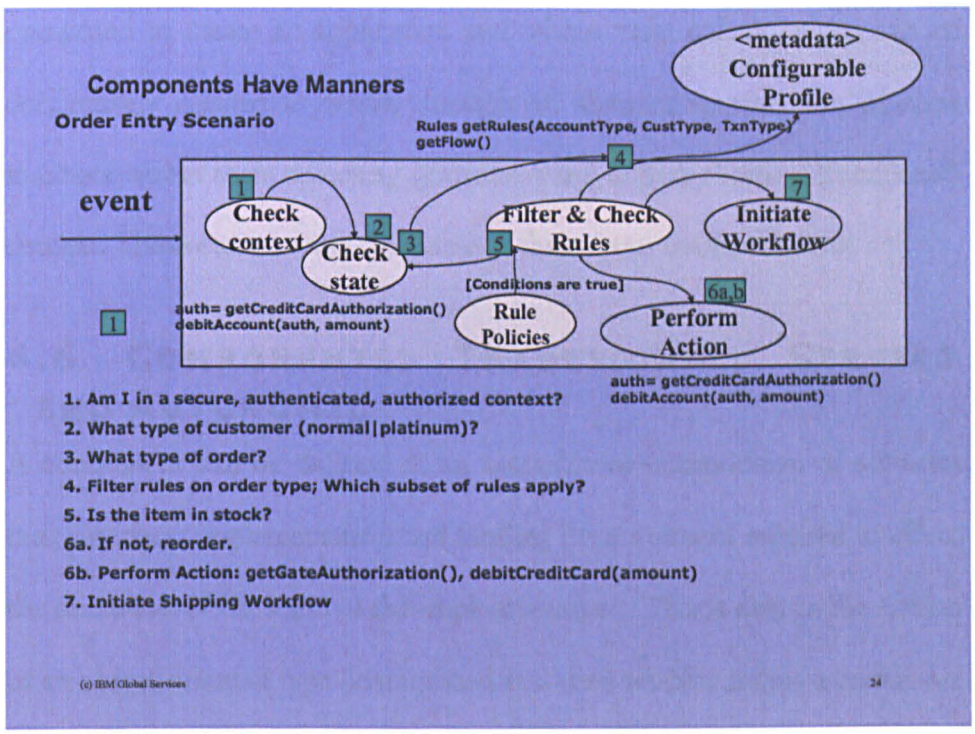


Figure 19: Manners Enable Context-aware Components

Once a CAC receives a message or responds to an event, first its context is checked, then its state is checked which is usually "session dependent". Then

the subset of rules that apply are filtered to a reasonable set that corresponds with the rules relevant to the given context. This is akin to the notion of manners in human life: rules govern the behavior appropriate to a given context. Then, if the rules apply, an action is taken or a workflow is initiated or an exception is generated.

4.5 Context-aware Components within a Service-oriented Architecture

We introduce the notion of context-aware software components that can be composed to create an application and whose valid collaborations can be declaratively defined *a priori*, through an abstract specification of their manners within their operating (infrastructure) and application (functional) context. These components can come in three types outlined below.

4.6 Components: Traditional, Service and Autonomic

A component can be defined as an assembly or composition of services; their interface, implementation and binding (to a protocol required to effect the execution of the logic of the implementation). This is akin to the notion of an object-oriented type (instantiated as a *class*) which exhibits a certain set of characteristic behavior. The behavior is often aggregated in the interface which defines the object type and becomes the method signatures of the class than implements the type.

In the world of J2EE, for example, a Java class can become an EJB by virtue of implementing the EJBHome and Remote interfaces and thus providing a set of standard method implementations that can be managed by the component container.

In the case of Entity Beans, for example, the container manages not only transactions, security, serialization but also persistence and caching.

Message-Driven components introduced in J2EE through the EJB 2.0 specification add the missing asynchronous element to enterprise components that can fire-and-forget a message, with the understanding that the underlying protocol on which JMS rides will guarantee the asynchronous delivery of the message request/notification.

The next step in the evolution of components came when it was realized that it is useful to distinguish component granularity:

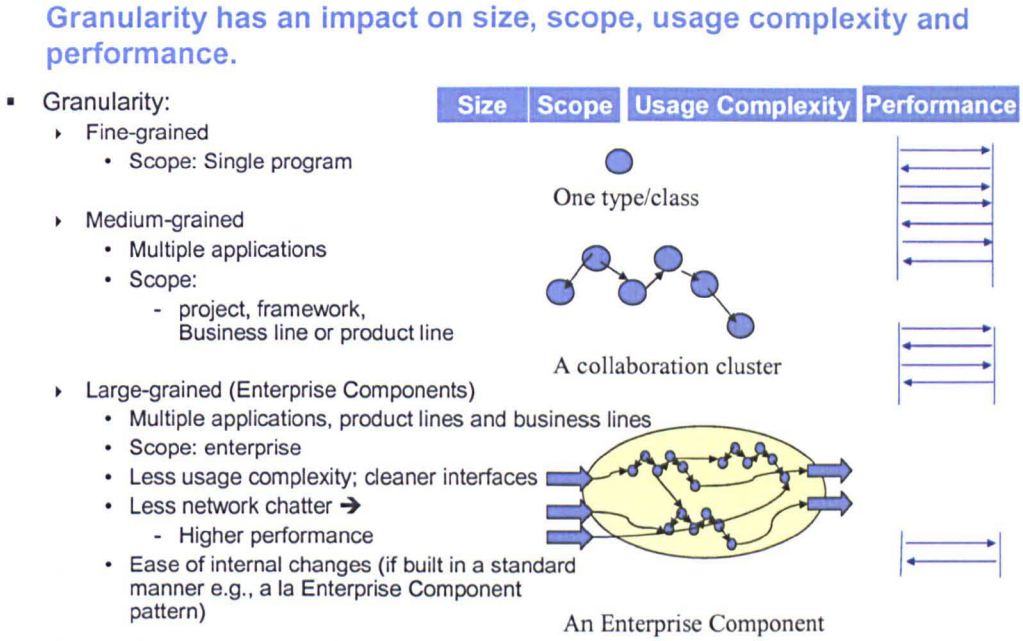


Figure 20: Granularity has an impact on size, scope, usage complexity and performance.

4.6.1 Web Services Composition

Similarly, from a web services perspective, the BPEL4WS standard defines a component composition model for Web services. Although not a formal component container, the aggregation or composition thus created needs to be managed by an implementation of the BPEL; for example in the BPEL4JEngine (see www.alphaworks.ibm.com).

Taking a step further and adding the following attributes to a component, we deem it "autonomic" [134]:

- Self-Configuration
- Self-optimizing
- Self-Healing
- Self-Protecting

The WS-Context and WS-Policy standardization efforts aim to address the space of defining extra-functional aspects of a WSDL service description which may be linked to policies that can be declaratively specified rather than hard-coded. This enables the realization of component and service manners.

4.7 THE GROUP CREATES CONTEXT

An application is created through the aggregation of a group of components -- often selected on the basis of the services they provide. However, in a transactional context, for example, components require knowledge of their mutual dependencies, on each other and on the system as a whole: their personal aspect and holistic aspect, in order to know which component's service to call next in the chain of collaborations that are on-going as a business process or computation unfolds. This grouping of objects or components itself creates a context which can be externalized as the group's context and a mediator class can be responsible for the instantiation of the collective, group based on the state of the group of components. For example, the shipment of a 1000 computers will require different mode of fulfillment (handling) than one computer, especially when there are

contracted quality of services involved that must guarantee on time shipment of all the batch of computers to a remote location.

4.8 Summary of the Notion of Manners

Manners are rules governing behavior of a reuse element (like a class or a subsystem) within a given context (usually, a business context).

Manners cover a wide spectrum, from business analysis expression of business rules assigned to a reuse element to a grammar-oriented object design specification based on a domain-specific language for a business. Here are some considerations regarding manners:

1. Business rules should become first-class constructs of the object and component paradigm.
2. Therefore, a class has not only identity, state and behavior, but also manners.
3. Manners are rules governing the behavior of a reuse element (class, ..., subsystem (--> component),...) within a given context, plus the meta-data needed to express the context, the rules and the conditions. Examples: XML, EJB Deployment descriptors, the Configurable Profile pattern (Part of the CBDI Pattern language) ["define a profile for each user type, so you can change the rules governing how the user type interacts with the system, their workflow, the rules that they have to abide by when they are offering products and services in a given office in a given

geographic region (context) for a corporate (customer type) client within these time frames (duration constraint).

4. Object discovery vs. component discovery - >fine-grained objects vs. medium to large-grained components. Instead of starting at a fine level of granularity, start with a higher level of reuse. Start your analysis at the subsystem level (a larger grained reuse element). Identify the manners for the business domain. What are the rules governing behavior in this business domain? What are the contexts in which business events occur? What rules /workflow should be triggered?

5. Write a domain-specific language to describe a business domain. This consists of a set of production rules. They can be externalized and represented as XML. This enables a subsystem's DSL to govern the laws of its business objects and how they interact and collaborate to fulfill a business purpose.

6. The Enterprise Component pattern is used to implement a subsystem whose manners were described using the DSL. The DSL essentially captures the workflow within the subsystem[* see point 6.3 below] . This workflow is managed by a Mediator within an Facade. The Mediator coordinates activities and workflow among a set of loosely coupled medium-grained components or plain business objects. The Mediator uses a Rule Object to externalize its workflow. Its colleagues (the components

or business objects; that's their name in the Mediator design pattern) may each have a Rule Object that governs their rules. IN this way, you can change a component's behavior at two levels in a non-intrusive way: change the workflow by changing the Mediator's Rule. Change the business component's Rules by adding new Rule Objects (Conditions or Actions). Thus, the Enterprise Component, which realizes a subsystem, will have realized its manners through the Mediator's workflow Rule Object. This can be represented as a grammar. This grammar (e.g, an XML DTD or Schema) can be parsed using standard XML parsing tools to decouple the workflow of the Mediator from its Colleagues (the business components or objects). - Thus, we have built an adaptive system with three key attributes: dynamic configuration, dynamic collaboration and self-description.

6.1- The ability to adaptively change the workflow without stopping the application is dynamic configuration.

6.2- The related ability of altering a set of collaborations between objects or components ""on-the-fly" is called dynamic collaboration.

6.3-Footnote: the subsystem grammar or manners, describes the rules governing the behavior of the subsystem under events and

invocations made to its facade. It also contains meta-data (the grammar) which can be queried (reflection) from another component who asks for our component to describe itself. This attribute is called self-description.

7. Thus, manners is a semantic notion of manners = rules + context + meta-data
8. Manners can be represented in a spectrum of realizations or implementation mechanisms:
   8.1. If-then-else
   8.2. Polymorphic Strategy-like implementation
   8.3. Rule Type = Rule Object + Type Object
   8.4. State Machine
   8.5. Grammar (this ties it in with the notion of a domain-specific language).

It is important to note that we must "pay as you go"; if we don't need the power, we must select a simpler implementation mechanism

4.8.1 Component Consciousness: Rules and Ontologies for Semantic Awareness of Usage Contexts

The degree of awareness a component displays with regard to its context and environment determines, to a large extent, the success of its behavior

within that context as well as its ability to fulfill requirements arising from being situated in the context.

4.8.2 Manners and Modularity

4.8.2.1 Formalizing the Notion of Modularity

The modularity of a component-based on-demand system is based on a set of elements that are considered members of the set. The modularity criteria can be expressed as a predicate $X = \{x \ isAMemberOf \ X \mid P(x)\}$. Parnas expressed a criteria for decomposition in which the modularity criteria is based on design decisions. The idea is to decouple design decisions that are likely

to change so that they can be changed independently. Design decisions can be formally defined by predicates (DDP's).

4.8.2.2 The Key design decisions

Component functionality F, or services exposed are a design decision.

Component service level agreements S, are a design decision.

Component manners M, are an additional constraint applied to a set of existing design decisions.

$X = \{ x \text{ isMemberOf } X \mid P(x) \text{ and } Q(x) \text{ or } R(x)\}$.

Component granularity G, is also a design decision.

The toplogy of a component is its ability to maintain its shape despite changes to its state and behavior.

Stability of a component combination is a tensor describing the stress on its formal manners specification.

The rapidly reconfigurable architectural style conductive to on-demand application composition and just-in-time integration is achieved through the notion of context-aware components, or components that have manners.

Manners is best described with the following illustrative example: assume you want to design a capability of a software banking (account management) component that transfers funds from one account to another.

4.9 AUTONOMIC COMPUTING AND MANNERS

A systematic view of computing modeled after a self-regulating biological system.

4.9.1 Agents and manners

Agents are autonomous objects that tend to be of smaller level of granularity than the enterprise components whose manners are cost-effective to externalize and manage. Agents can exist within a larger enterprise component boundary and support the runtime functionality of the internal operations of such an enterprise component.

4.9.2 Components and Manners

Needless to say, that in all of the above cases, the underlying theme is that the behavior of the component, whether an agent, an autonomic component or an object, needs to be specific in an abstract manner and yet be executable, to enable the mapping from a specification level to an implementation level.

This behavior is contingent upon the context, state, policies, rules that exist and are externalized as manners.

Thus, the addition of manners to any component makes them effectively a context-aware component.

5 Chapter Five: Implementing Executable Business Specifications using GOOD

Objectives

- To describe how to create executable business process specifications using Grammar-oriented object design

- Discuss the specification, design and implementation of a tool implementing Grammar-oriented Object Design

5.1 Semi-formal, Configurable and Executable Business Specifications

The notion of a precise business specification that is precise enough to be used to generate or execute business flow within a business application was a delicate balance of formalism and the constraints of realism imposed by pragmatic necessity: deliver artifacts that are not just intellectual exercises but that add value to the design and implementation of business systems projects.

One of the first hurdles was to introduce the notion of business modeling into the software development life cycle. As late as 2000, business modeling was not a formal part of the industry de facto standard, the Unified Software Development Process (initially Rational Unified Process).

Although there are various business process modeling methods, Jacobson was the first to utilize object-oriented notions to perform business engineering [48]. Eriksonn and Penker were the first to publish a treatise on the utilization of UML, business patterns and marry this with business modeling [35]. This empowers the business analyst to specify their often highly ambiguous business requirements and functional requirements with a degree of accuracy that the software architect can help in developing as well as carrying to fruition.

This fusion of the notions of domain-specific languages, semi-formalism specifications with component-based software engineering and object-oriented analysis and design with an emphasis on adaptive software architectures is called Grammar-Oriented Object Design (GOOD).

If specifications are not executable then above discussion points towards what is probably the closest approximation that can be made.

5.2 TOOL ARCHITECTURE

The architecture used the GOOD Dynamic Controller architecture:

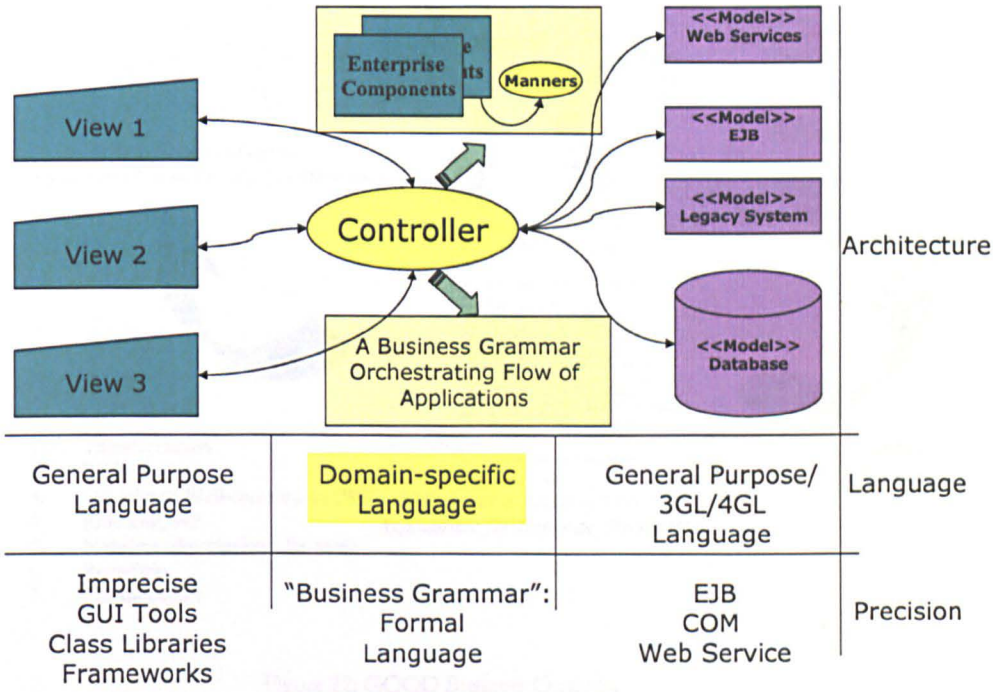


Figure 21: GOOD Dynamic Controller Architecture

The controller used a special scanner that read from the input stream, events or using specialized tokens, an LL(1) Parser, a code generator and an action service registry.

There is a standalone and integrated version of the Business Compiler that was developed.

5.2.1 Internal Representation with Grammars and Input Mechanisms

The Business Compiler is a tool that implements GOOD.

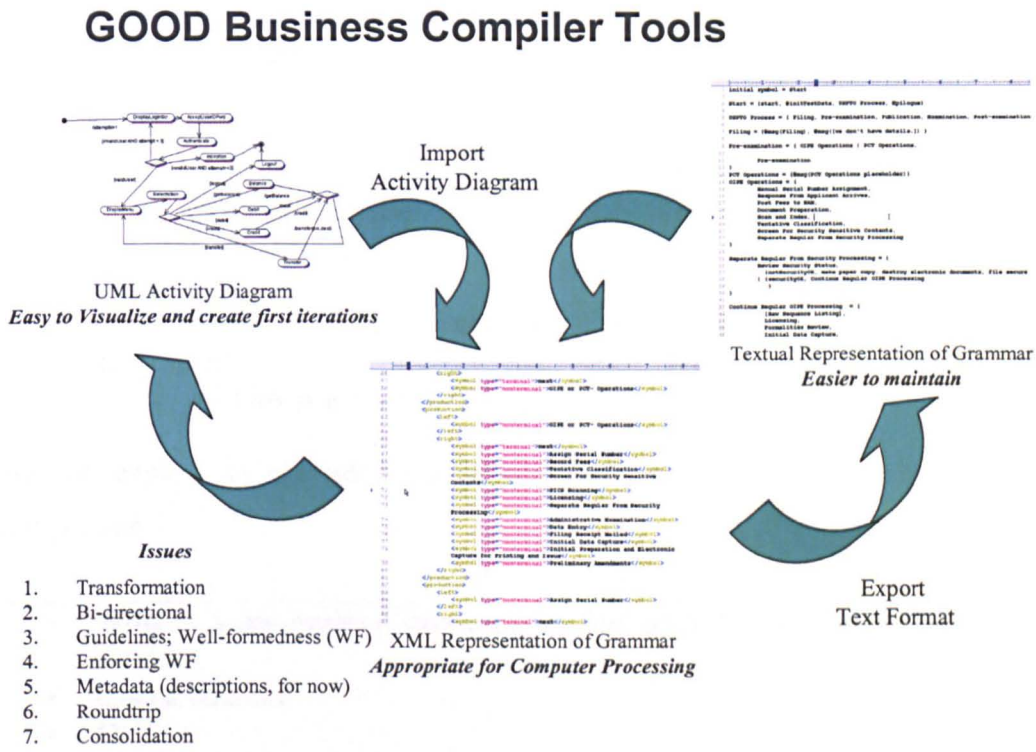


Figure 22: GOOD Business Compiler

The GOOD Business Compiler takes two types of input, namely, a Rational ROSE activity diagram can be imported and a business grammar (manners) generated. Alternatively, the business grammar can be input via a text editor. The intermediate interpreted representation is in an XML standard that was developed for this purpose.

After extensive industrial parser evaluation it was determined that there were too many dependencies inherent in the public domain and commercial parsers. Thus an LL(1) parser was written to serve as the parser/interpreter for the business grammar.

5.2.2 Syntax of the Business Grammar

The syntax of the business grammar is a standard LL(1) syntax which can be described as follows:

- *Sequence*
  - *Process = {DoThisFIrst, DoThisSecond, DoThisThird}*
  - **Patent Process = { Filing, Pre-examination, Publication, Examination, Post-examination }**
- *Conditionals, Branching*
  - *Process = {DothisFirst | YouMayDoThisFirst, DoThisSecond}*
  - **Pre-examination = { Initial Operations | PCT Operations}**
- *Loops*
  - **Txns = { displayMenu, Txn, Txns} | doc received**
- *Comments*
  - **//This is a comment**

We will explore an example of a grammar for an account management component.

- **Write a large-grained component that provides the following services**
- **Check balance**
- **Debit**
- **Credit**
- **The component requires**
- **Access to legacy systems for database and systems of record**
- **Rules**
- **An Account can be Savings or Checking to start with**
- **Customers are either Regular or Platinum**
- **If a Customer is Platinum and they overdraw, then they can get an automatic loan based on their credit worthiness (requires a service called CheckCredit(ssn))**

We now need to perform domain decomposition according to the method and write a business domain grammar for the account management component. This can be done in two ways, a direct entry in a text editor or through its specification via an activity diagram.

Here is the grammar that was written for this purpose using a text editor.

```
Account Mgt = {Login, Txns, Logout}
Login = {# displayLoginPage, login, # getUserIdAndPassword, # login,
CheckLoginResult}
CheckLoginResult = {{success, DisplayMenu}
| {invalidUseridOrPassword, # displayInvalidUserError, Login}
| newError, # displayNewERRor, Login| {error, # displayError,Login}
}
DisplayMenu = {displayMenu, # displayMenu, # getTxnType}
Txns = {Txn, DisplayMenu, Txns} | end
Txn = {{accountInfo, AccountInfo}
| {debit, Debit}
| {credit, Credit}
| {transfer, Transfer}
}
AccountInfo = {getAccount, # getAccount, # displayAccountInfo(account)}
Credit = {getCreditParameters, # getCreditParameters,
# performCredit(srcAccount, amount)}
Debit = {getDebitParameters, # getDebitParameters, # getCustomerType,
CheckFund, CheckFundResult}

CheckFundResult = {
{success, # performDebit(srcAccount, amount), CheckTxnResult}
 | {invalidAmount, # displayAmtError}
| {insufficientFund, # displayFundError}
| {error, # displayGeneralError}
}
CheckFund = {regularCustomer, # checkFund(srcAccount, amount)
| platinumCustomer, # checkCredit(srcAccount.getBalance()-amount)}
CheckTxnResult = {
{success, # displayDebitConfirmation}
| {invalidDebitAmount, # displayDebitAmtError}
| {error, #  displayGeneralError}
}

Transfer = {
# getTransferParameters,
```

```
# determineCustomerType,
CheckFund,
# performDebit(srcAccount, amount),
# performCredit(srcAccount, amount),
Confirmation
}
This involves CheckFund, CheckFundResult
CheckFund = {
regularCustomer, # checkFund(srcAccount, amount)
| platinumCustomer, # checkCredit(srcAccount.getBalance()-amount)
}
```

Alternatively, here, in Figure 23, is a rendition using Rational ROSE of the flow of the system using an activity diagram in UML notation.

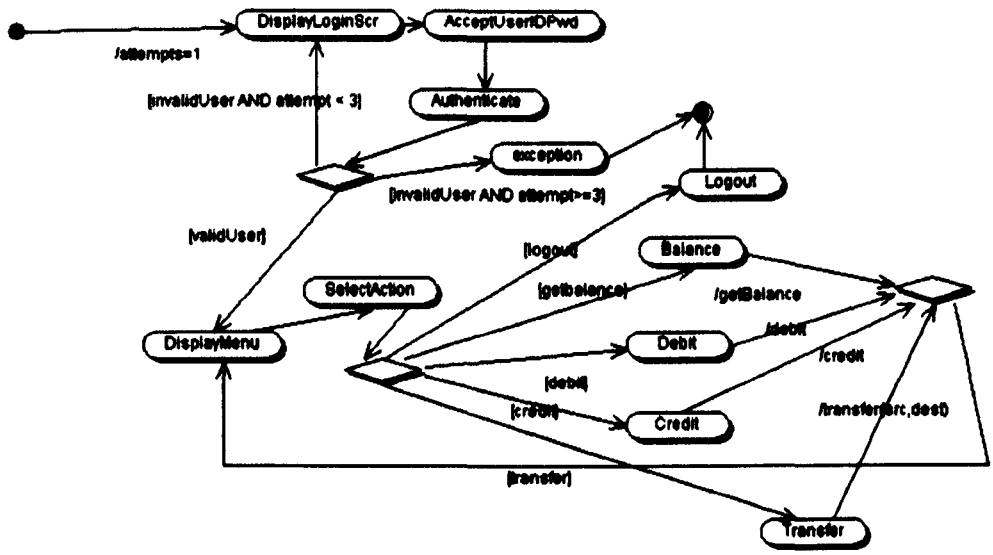


Figure 23: Activity Diagram as input into
Business Compiler Tool

A sample user-interface for the Business Compiler may look like the following:

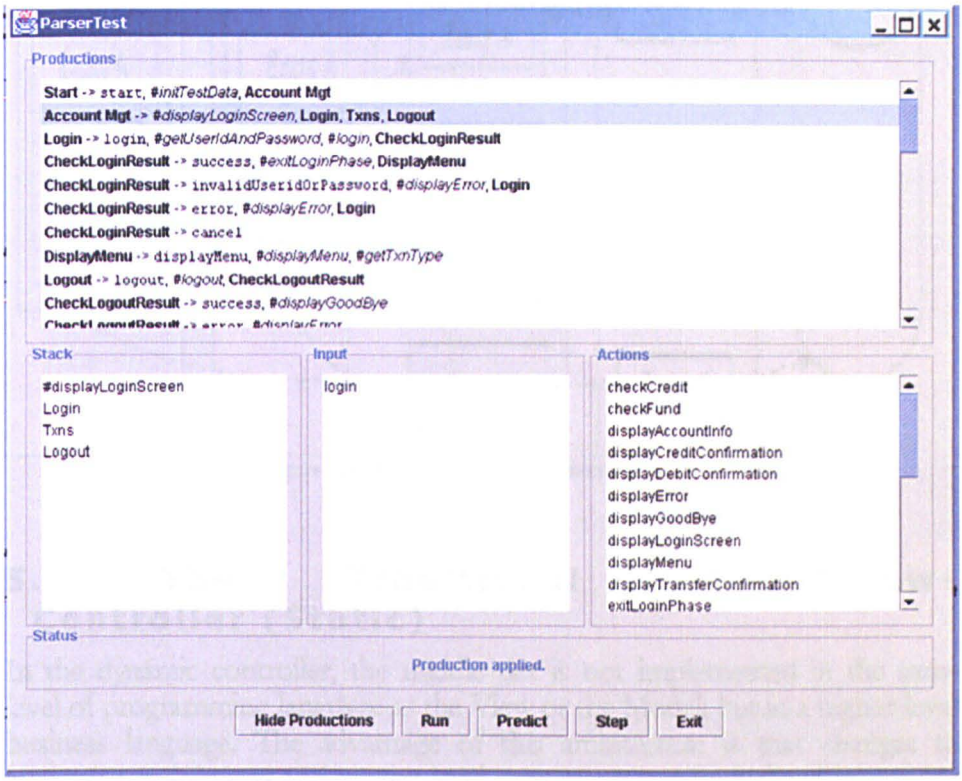


Figure 24: Sample user-Interface for GOOD Business Compiler

The Business Compiler outlined above is an implementation of a Dynamic Controller Architecture. A Static Model-View-Controller architecture outlined below.

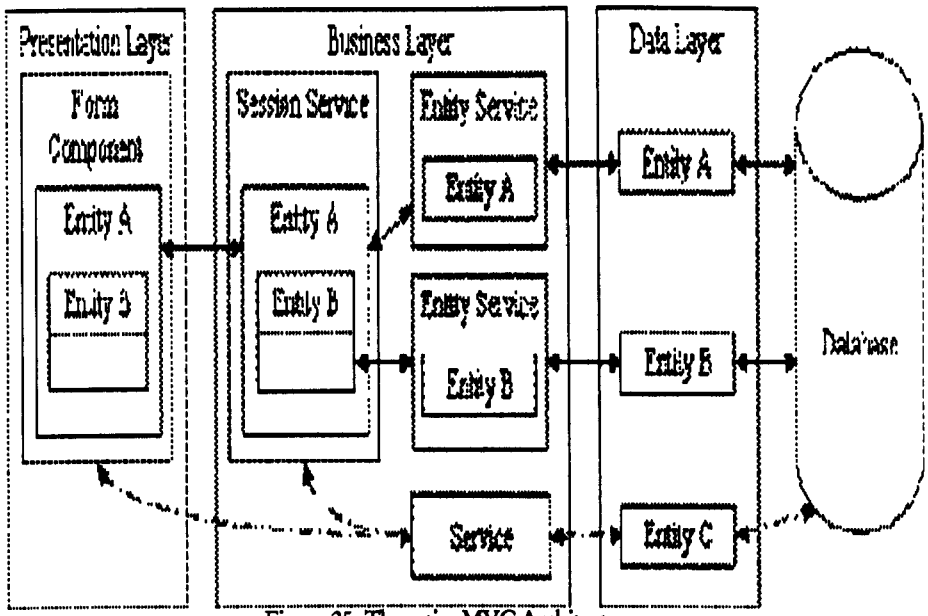


Figure 25: Three tier MVC Architecture

5.2.3 The Traditional Model-View-Controller (Static)

In the dynamic controller, the middle tier is not implemented in the same level of programming language as the View or the Model; but in a higher-level business language. The advantage of this architecture is that changes to business requirements and service level agreements can be realized by making configuration changes to the dynamical controller which is in-line with business specifications.

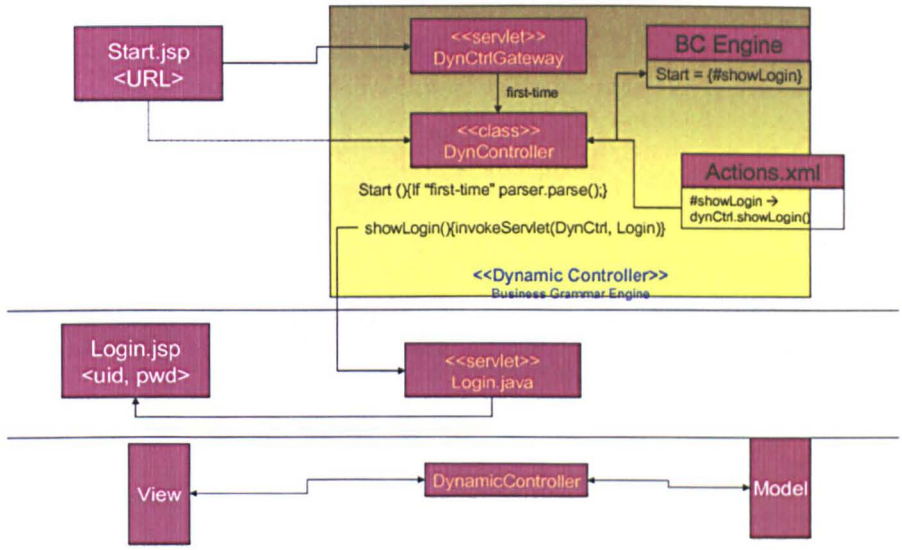


Figure 26: Dynamic Controller Sample Architecture N-tiered Realization

The dynamic controller architecture uses the implementation of GOOD in the form of the Business Compiler Engine. This serves as the dynamic controller of the architecture.

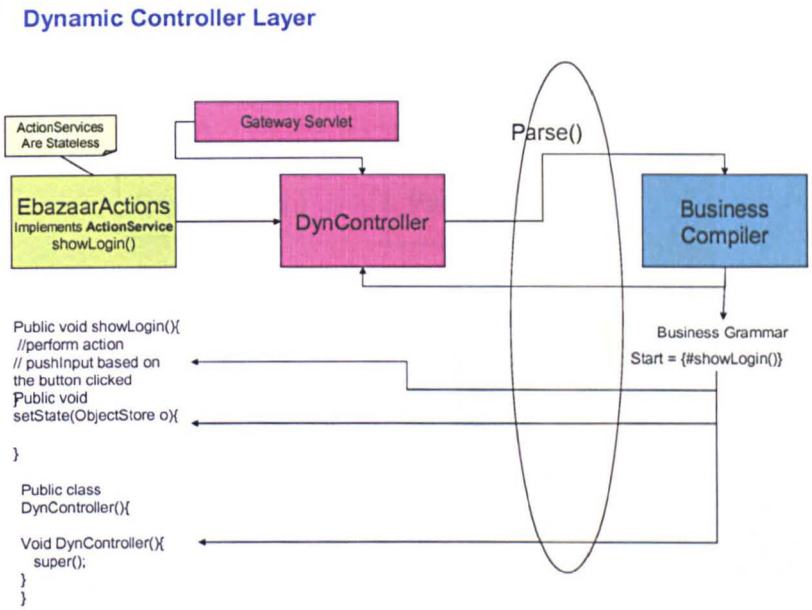


Figure 27: Dynamic Controller Layer

Figure 27 shows the dynamic controller layer, showing the relationship between actions, the layer that is the dynamic controller (versus a static controller) and the Business Compiler Engine.

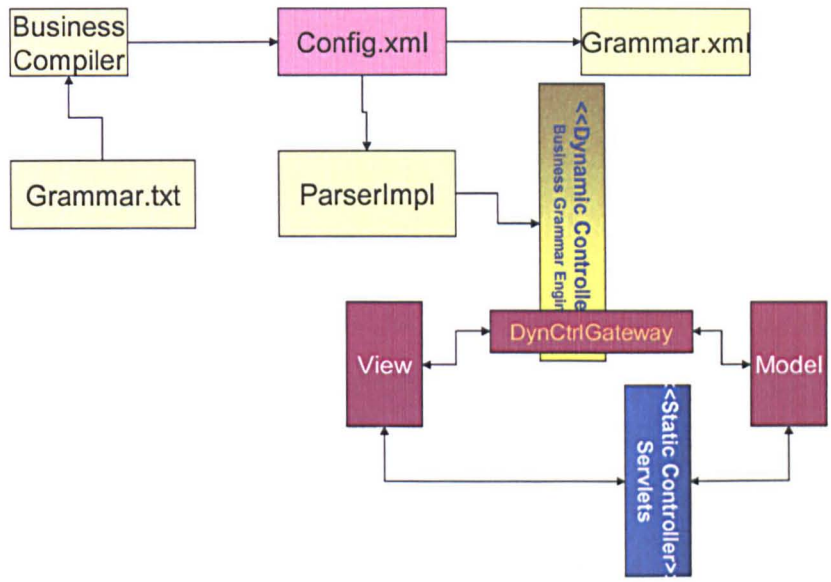


Figure 28: The Design of the realization of the Dynamic Controller Architecture

5.2.4 Key Considerations

- *There is a Business Architecture in place or needs to be defined*
- *Requirement is to allow the business modelers to define and represent their own business process flows, without heavy dependence on IT turnaround times*
- *And yet you need a degree of precision to create prototypes, production systems and software in general that will support the business being modeled.*
- *Modelers have a strong prior knowledge or their business domain;*
  - ... a subject matter expertise that is invaluable to the successful execution of a software development project that is designed to support the business.

- *We therefore use grammar-oriented object design (GOOD) to create a formal specification of the business domain and execute it using a Business Compiler*

6 CHAPTER SIX: IMPACT AND IMPLICATIONS OF GOOD

Objectives

This chapter discusses the relationships, impact and implications of GOOD on software architecture, business architecture

- To explore the specific implications of GOOD for Service-oriented Architecture and Web Services

- To explore the impact of GOOD in shaping on-demand computing at a functional level

6.1 Impact area One: Software Architecture

6.1.1 Implications of GOOD for Software Architecture

The categorization of rules, components and workflow into three levels of granularity is not a coincidence. There are multi-dimensional separations of concerns [98] that are addressed by this distinction of levels.

A dynamically re-configurable architecture is defined as an architectural style that is configurable to meet functional needs and tunable to be meeting non-functional requirements through enabling optimizations necessary to fulfill service level agreements to ensure quality of service [14]. Yaghoub, et al., (2000) describe some of these optimizations for data-intensive web-based systems [97].

In order to accommodate these configurations and optimizations, variant aspects of the software architecture must be identified as early as possible [12, 14], encapsulated, separated out of the main stream of application or business logic that conducts day-to-day operations of the business processes or the functioning of the real-time system, and finally be externalized to provide axes of configurability. Arsanjani, et al. [12] describe the process of conducting these externalizations of such properties in a Configurable Profile.

6.1.2 Architectural Styles

The next domain that resides at the junction point is that of software architecture and its associated styles. Garlan and Shaw [68] have been the initial pioneers in this field.

Software architectural styles are key design idioms [8, 24]. UNIX's pipe-and-filter style is more than twenty years old; black-board architectures have long been common in AI applications. User interface software has typically made use of two primary runtime architectures: the client-server style (as exemplified by X windows) and the call-back model, a control model in which application functions are invoked under the control of the user interface. Also well known is the model-view-controller (MVC) style [15], which is commonly exploited in Smalltalk applications. This architectural style is more recent, and has an associated meta-model [38].

The architectural problems encountered in organizations today stem from a rigid architectural style that was introduced in a random fashion as the application systems grew and evolved. The result of each change was a more brittle architecture. Enhancements to enterprise applications that typically focus their domain within a product line or business line typically introduce more entropy into the already saturated structure of the architecture. Having not been designed for change, with technologies that provided very little flexibility has resulted in IT application architecture not being able to support the often changing and rapidly evolving requirements of business goals, drivers and processes.

The notion of decomposition [73] and separation of concerns [99] are key notions common across architectural styles. For a highly-reconfigurable and adaptive architecture, these notions gain prominence and become critical. Subdivision of system functionality into components is essentially the traditional problem of modular decomposition. As such, components which encapsulate a cohesive set of features or function-points, or which encapsulate functionality likely to change, are appropriate candidate components, and would be appropriate modules in any style. In particular, a message-based component architectural style affords good separation of concerns by supporting implicit invocation via notification.

Thus the key characteristics of a strong component-based architectural style are:

- Clear separation of (often multi-dimensional) of concerns
- Loose coupling
- Externalized coupling control
- The ability to control the coupling and flow of component collaborations without significant impact on the operation, functionality or performance characteristics of components.
- Encapsulation of a set of cohesively related changing aspects or concerns

- Layering: only higher level components are visible to any component and the dependencies are not on actual lower level components
- Adaptability or configurability: The ability to rapidly reconfigure a component configuration or a component itself to be relevant and Pluggable within a given context.

The notion of highly-reconfigurable architectures comes into play with the notion of services on demand as the need for dynamically composing application and infrastructure components becomes apparent within the realm of services on demand, in a utility-based model of computing. Each large-grained component knows its manners and the valid ways of interacting with its environment.

Rules build manners which build self-awareness and context-awareness in autonomous components, enabling them to be combined in valid permutations on an as-need basis. This highly re-configurable software architecture describes the manners of components through GOOD. Languages can be used to describe architecture in general and a HRA/DyRec in particular.

6.1.3 Architecture Description Languages

Architecture Description Languages (ADL) describes the structure and construction of software architecture. In this sense they are akin to connector and port definition of components as well as the notions of a

component composition or assembly of components. These specialized languages provide a mechanism for a more formal description of software architecture.

6.2 Impact area two: Implications of GOOD for Service-oriented Architecture and On-demand Computing

6.2.1 Service-oriented Architecture (SOA)

This architectural style is based on the even stronger separation of interface and implementation than that presented in object-oriented programming. A Service Provider creates a Service and Publishes it to a Service Broker who will provide a Service Discovery mechanism (such as UDDI, Uniform Description, Discovery and Integration standard). Then a third role, the Service Consumer will Find a Service using the Service Broker's Service Discovery mechanism. Once found, the Service is Bound to and directly invoked in a peer-to-peer fashion from Service Consumer directly to Service Provider, as simply as you would click on and open an HTML link to bring forth its contents. This Triad of Producer, Consumer and Broker form the three roles of an SOA.

This style has been implemented (starting in 2000) using the notion of Web services.

6.2.2 Separation of Representation from Presentation

The Internet gave access to documents. The next step in the evolution of this paradigm of inter-connectedness was to provide not only access to information (static), but to services (dynamic) able to be invoked over the Internet. In order to be able to invoke a service over the Internet, a new programming and modeling paradigm was needed to augment the notion of Remote procedure Calls (RPC) and take into account the recent developments in standardization of messages through the eXtensible Markup Language (XML) and J2EE technologies.

The notion of service-oriented architecture was introduced that provided a set of standards such as SOAP (Simple Object Access Protocol) , WSDL (Web Services Description Language) and registries (UDDI, Uniform Description, Discovery Interface) to discover available services. Using the more involved and complex SGML as its basis, but simplifying notions which led everyone away form SGML but a selected community, and focusing on e-business concepts while creating a viable and marketable subset called XML proved to be a solution to a long standing problem of separating the representation of data from its presentation (via HTML or WML for example, based on the needs of a given device the user was utilizing). Not only being able to "define your own tags" but to base electronic interchange of information content for specific industries and

applications started a trend of standardization that has been since ongoing (www.oasis.org) .

6.2.3 Web Services

Web services are the interfaces that are exposed in a distributed, application to application interaction across the World Wide Web (http://www.w3.org/2002/ws/). They are a realization or instantiation of an SOA, using technologies and standards such as WSDL (Web Services Description Language), SOAP (Simple Object Access Protocol), UDDI and XML (extensible Markup Language). Various other standards are being completed.

6.2.4 On-demand Computing

On-demand computing refers to the ability of a computing environment to dynamically respond to changes in its environment or requirements and respond to the threat or opportunity that has arisen. On-demand computing has been defined as an environment whose attributes are Open (based on open standards versus proprietary technologies), Virtualized (based on the grid services such OGSI (Open grid Services Architecture)) [144], Integrated (application within and between partners in a value chain are integrated seamlessly end-to-end) and Autonomic (the infrastructure and computing environment has the ability to adapt to failures and self-heal, self-configure, self-optimize and self-describe itself).

Other models have been described as the Enterprise Nervous System, the Dynamic Enterprise, Just-in-time integration, etc.

This is reminiscent of the SEI Capability Maturity Model level 5, optimized, where feedback loops provide input into a statistically self-regulating system. Arsanjani outlines an Integration Maturity Model (IMM) [15] that shows how integration can be taken to higher levels to achieve this ideal of dynamic realization of business needs required by on-demand computing.

6.2.5 A Utility-based Model of Computing

Providing information technology, applications and infrastructure as a utility is not a new concept. It's realization; however appears to be more and more realistic and economical.

A variable cost structure defines a "pay as you go" model where the client pays only for the amount of metered services rendered.

This is similar to the Application Service Provider or ASP model where software functionality is provided in a metered fashion, much like electricity or water: a software package is not purchased but made accessible for usage on a fee for unit-of-usage basis, such as fee per transaction, or volume, "number of clicks" or customized reporting or billing capabilities.

Figure 17 shows a maturity model for the levels of increasing integration of services. This implies an incremental increase in the integration capabilities

of the applications and business at each level. To take the quantum leap into the next phase or level, an organization must undergo a transformation or evolution (drastic or steady depending on its current state).

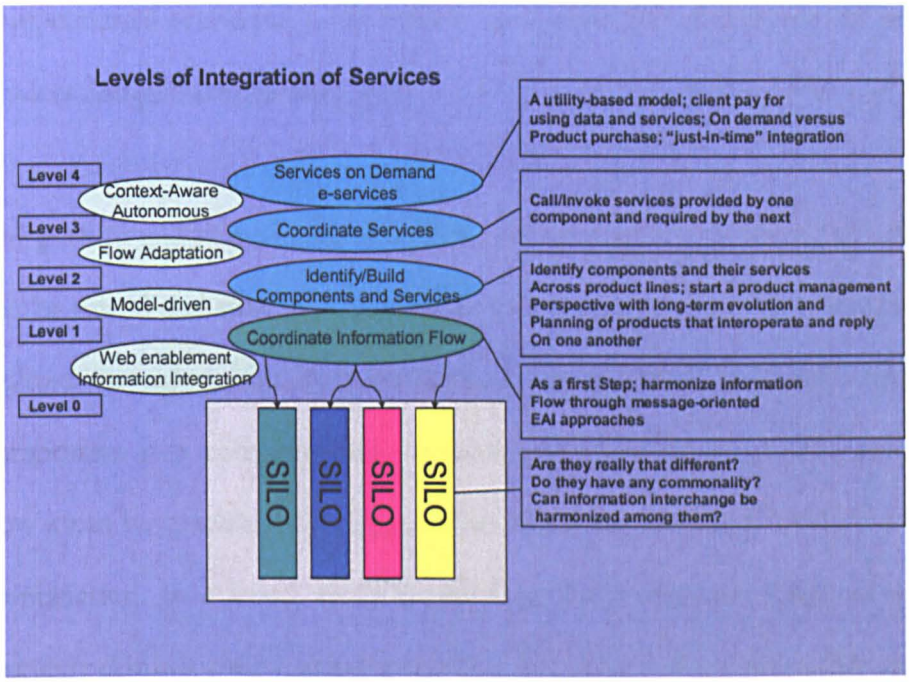


Figure 29: Levels of Integration

Utility-based computing is an aspect of on-demand computing. In order to realize utility-based computing, a standard mechanism for its large-scale creation and consistent deployment is needed. Grid computing provides such a standards and technology basis. Grid Computing is emerging as a next-generation parallel and distributed computing platform driven by the Internet, Web services technologies, and service-oriented computing architectures. Grids enable the sharing, selection, and aggregation of geographically distributed resources, such as computers (PCs, workstations,

clusters, supercomputers), data sources, and scientific instruments, for solving large-scale problems in science, engineering, and commerce. Thus, the word "Grid" has come to denote middleware infrastructure, tools, and applications concerned with integrating geographically distributed computational resources. Grid services is a powerful combination of web services and grid computing [136].

The path to realizing a highly decoupled set of interacting components that can be mixed and matched dynamically to form applications on demand is realized through grammar-oriented object design. Each semi-autonomous component is a context-aware, context-sensitive component that knows how it can be combined through understanding its own set of valid rules of combination. In a word, these components have manners. They behave correctly through their context-awareness and can combine with other such self-aware/context-aware components to produce a type of complex adaptive systems that we call dynamic applications.

6.2.6 Grammars and Formal Definition of Document Structure

XML DTDs brought the notion of grammars back to the forefront[5]; defining a document's structure by using elements, attributes and entities. This evolved into the XML Schema specification.

[5] Although SGML had pioneered this, it had not gained much adoption.cx

XML was extended and used for various aspects of information interchange (validating and non-validating XML parsers, Document Type Definitions and XML Schema [OASIS]), remote invocation (Simple Object Access Protocol [MS-IBM]) definition of services via Web Service Description Language (WSDL), XPath, Xforms, XSL (XML Style Sheets) and XSLT (XSL Transformation) and WSIA (Web Services for Interactive Applications) [Arsanjani et al], etc., were introduced.

Thus, using XML to transact business-to-business interactions abounded: ebXML (e-business markup language), TPML (trading Partner Markup Language).

6.3 Impact Area Three: The Implications on Business Modeling and Business Architecture

Although current object-oriented analysis and design methods and modeling languages have each a rich set of semantics and syntax to support the modeling of software systems, their support for business modeling has been more limited. In addition, there is no full life-cycle support for the challenges posed by component-based development and integration (CBDI). The transition and traceability from business models to software component architectures is by no means smooth. This paper explores extensions to current methods to fully support CBDI across the life cycle starting from business modeling and throughout the software development lifecycle.

We propose Grammar-oriented Object Design (GOOD) as a method of identifying and mapping reusable subsystems in a business model to a well-mannered component-first software architecture. The latter architecture assumes the modeling and design process is based on an assembly-based paradigm of wiring pre-built or customized components that have been designed for change and with the intent of repeated long-term reuse and customization.

6.3.1 Methods and Business Modeling

Current object-oriented methods have been augmented to include work products and activities for business modeling or have "standardized" extensions [144]. Examples of these methods include the Rational Unified Process [112], the Unified Modeling Language (UML) [84] and the IBM Global Services Method [47]. Business Extensions for the UML, the Business Domain in Global Services Method and the Eriksson-Penker extensions to the UML for business modeling [35] directly support business modeling. Business Modeling supports software modeling by identifying the information systems that best support the operations of the business [111], serving as a basis for functional and non-functional requirements, and as a basis of refinement into analysis and design, as well as, providing a basis for identifying suitable components [19].

The transition from business modeling to software modeling is not a smooth or well-understood process. Often, different teams that have different

terminologies, diagramming, visualization techniques and goals perform the activities of business and software modeling. Thus a way to seamlessly map these onto one another is required.

6.3.2 Business Architecture and Enterprise-scale Components

There are many definitions and interpretations of what is signified by component-based development and integration (CBDI) and what defines a "component" [23]. A software component is "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties" [25]. To fully realize component-ware, we propose an operational definition of how to actually build a technology-neutral components using architectural patterns, based on the need for components to be elevated from small-grained objects to cover a richer spectrum of granularity up to the enterprise-level business process. These large-grained enterprise-scale components can be wired together and adapted in a modular fashion to incorporate new behavior and business rules across business lines and product lines.

We define an Enterprise Component (EC) as a compound design pattern that will allow design-time specification of technology-neutral components whose realization mechanism can be based on any standard run-time component models such as EJB or .NET. An EC is a compound pattern consisting of a Composite Mediating Façade with "pluggable" Rule Objects [15], that is,

business rules that can be adaptively plugged into a component business model without violating the open-closed principle.

6.3.3 Business Modeling

A complete architectural specification of an information technology system includes information about how it is partitioned and how the parts are interrelated. It also contains information about what it should do and the purpose it must serve in the business [19]. One of today's foremost software development challenges is the need to balance business demands for faster time-to-market with a relatively volatile set of requirements on the one hand, with the need to deliver quality, reusable code. This challenge is becoming overwhelming to software development organizations. It calls for various techniques such as ensuring adherence to Meyer's open-closed principle [66], using adaptive object modeling [11] [143], and designing components with pluggable business rules [7]. All approaches nevertheless require strong business requirements based on an understanding gained through business modeling. The prevalent tacit expectation seems to be that during software modeling, the gap in understanding of business needs, rules, goals and perspectives will somehow be fulfilled.

It has been seen that producing business models provides a sound basis for creating suitable information systems that support the business [35]. Electronic business systems have the most to gain from an assembly-based paradigm founded on component-based development. Unfortunately,

balancing the need for rapid time-to-market with quality has not been easily or repeatedly achievable with the prevalent levels of ambiguity in business requirements (usually with little or no significant business modeling) and the indeterminate way they are mapped to the software architecture. This mapping will simply not work for component construction that demands partitioning of a domain into subsystems with well-defined manners. The business analyst normally bases business requirements on the knowledge the domain expert captured, but this approach does not guarantee precision or traceability. Software engineers are thus left to exercise unnecessary creativity in an area in which they are not well conversant – the business domain. This, in turn, leads to entropy in the software architecture, yielding mismatched expectations and systems that do not fit business needs.

6.3.4 Software Modeling and Architecture

Optimally, software is modeled by taking the business requirements that are an output of business modeling and performing analysis and design activities on the solid basis of correct and prioritized business needs [30][35][47]. We propose explicitly basing the initial modeling on domain-partitioned subsystems that map to distinct business domain such as Account Management, Customer Relationship Management, Rating, Billing, Product Management, Procurement and Shipping. Subsequently we suggest discovering the business rules and constraints governing the behavior of the domain plus the meta-data required to reflectively query the component for rules and services at run-time; i.e., the subsystem manners. The manners are

represented as a business domain-specific language that can be adaptively modified at run-time [110].

7 Chapter Seven: Method Support for the Design and Implementation of Reconfigurable Component-based, Service-oriented Systems

Objectives

- To provide key activities and artifacts that support the design and implementation of component-based and service-oriented systems

Terminology

The following diagram summarizes the key notions in this method in terms of the differentiation of business and IT terminology.

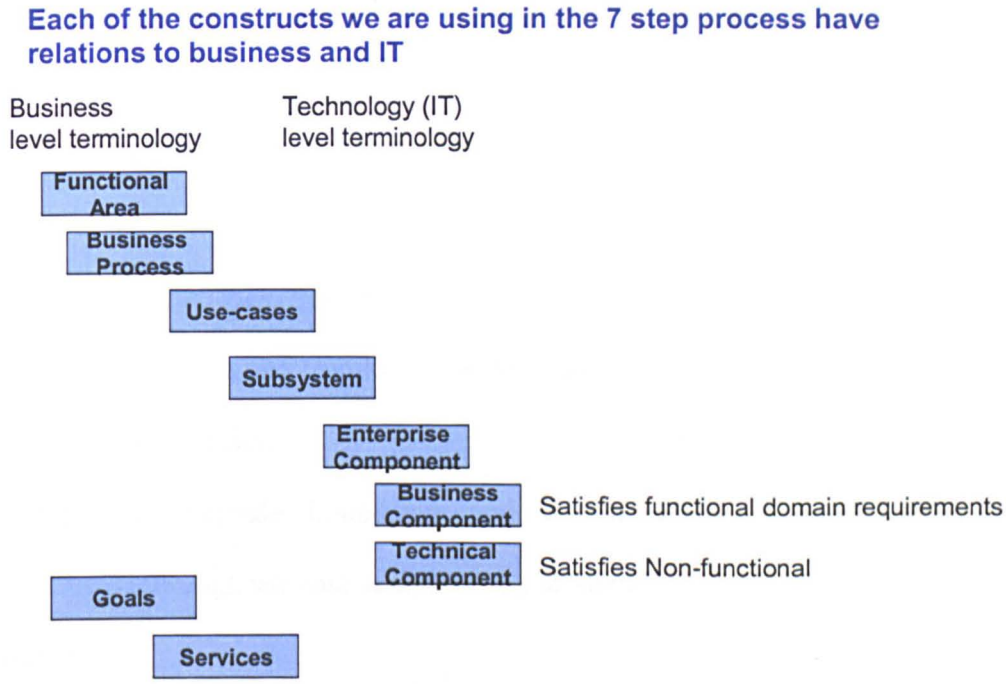


Figure 30: Business and IT Terminology

7.1 Extensions to Current Methods for Component-based Software Engineering

The following chapter describes the artifacts and activities that extend current object-oriented methods for performing component-based software engineering.

As outlined in [14] and in [12], we adopt a combination of a top-down and bottom up approach to component-based service-oriented system design.

The top-down aspect comes from taking business perspectives and models into consideration: business function, process, sub-process, business use-cases are elaborated and form the outlines of component boundaries.
This creates a blueprint for a component domain model; allocating conceptual containers for which to categorize and later actually use to implement services and chunks of business functionality often enunciated in the form of use-cases.
Components provide boundaries and containers for services (often discovered through use-case analysis and goal service model creation [14]) to reside in.

7.1.1 The Layers of Service-oriented Computing and Architecture

Consider the following Figure 31 as a map describing a conceptual view of the end-state of an SOA.

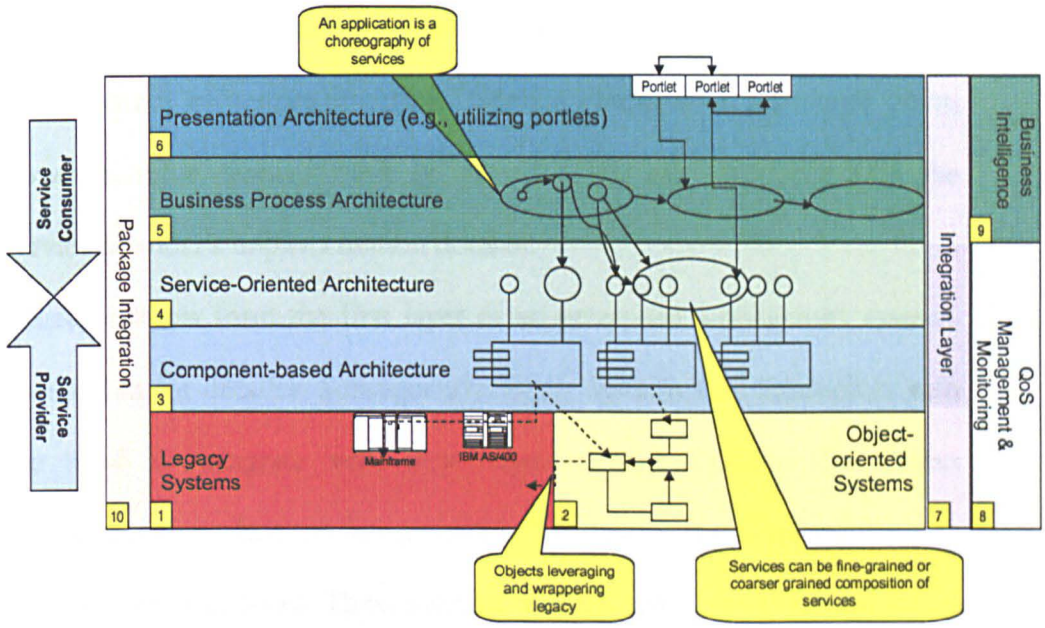


Figure 31: Layers of a Service-oriented Computing Architecture supporting Services on Request

A SOA is not one component or technology; it is more of an *approach* to designing a layered enterprise architecture consisting of multiple infrastructure and application components that rely on a set of loosely-coupled interfaces called service descriptions.

An SOA builds on several layers of the architecture as shown in Figure 31. An SOA often composed of services or components provided by the

underlying (often finer-grained) layer of services. The set of services that is available to be composed into a Business Process Architecture through mechanism such as BPEL, used to create an orchestration of services, is founded upon a layer of exposed Services. This layer of Service Architecture provides (functional perspective) and supports (non-functional perspective) the exposure of service interfaces (service descriptions. At some point, however, these services must be provided by some component in the service provider's implementation domain.

Legacy systems form the first layer of an enterprise architecture; systems invested in for decades. Subsequently, legacy systems find themselves with the need to integrate with new object-oriented systems (2). Object architectures are often too fine grained to be of any real use in a value-net in a cross enterprise sense. These systems are thus often re-partitioned into a more coherent, organized and reusable set of enterprise component (3). These components contain various business and technical components that provide functionality to support the business. In mining out the true business level of functionality from system required functionality, services can be defined and exposed at the boundary of these components and combined with services newly discovered or created to form the fourth layer (4) in the SOC architecture. Note that these services can themselves be composite: i.e., consist of other combination of finer-grained services.

These services can leverage packaged applications (10) that expose their services through more than merely API's, although API's can be leveraged at the legacy, object or component architecture layers.

Layer (5) is created by combining these services into composite services that do not merely aggregate functionality ("subsuming": "the big fish eating the smaller fish") but do so in a collaborative (peer to peer) fashion using orchestration as their key composition mechanism. This differentiating factor allows the creation of goal-oriented business processes.

These business processes support transactions that ultimately require presentation to a human or to a virtual interface ultimately traceable by a human. This Presentation Level (6) may be composed of any mechanism. However, portlets and portal technology seems to be the direction that most enables combination and re-combination of services that have a "face". A good example of a standard that supports this is Web Services for Remote Portlets (WSRP) and the Web Services Experience Language (WSIA).

In many financial applications as in many MIS or EIS (management or executive information) systems, the need for Business Intelligence (BI; layer 9) comes into play and helps decision makers monitor and make decisions about the running of the business. Thus, the need for more seamless integration of BI into the service-oriented computing framework arises.

The integration of all the disparate layers can be accomplished through a Service Bus supplied by the Integration Layer (7).

At the cornerstone of services lies the implementation technology which needs to be monitored and metered to ensure and enforce quality of service (layer 8).

In many cases the usage scenarios described in section 3.4 above enables the leveraging of a layer in the architecture to enable new application functionality and integration.

The above figure describes an end-state. How do we achieve this end state? There are four major approaches that can be taken to achieve this in a stepwise fashion as described in 4.5.2. Often, we will see it is prudent to combine steps. This combination is shown in section 4.5.2. Then the method for SOA is described in 4.5.3.

7.2 THE FOUR MAJOR APPROACHES TO SOA

There are generally four ways to approach SOA as depicted in Figure 32: Four Approaches to SOA : top-down, bottom-up, totally Greenfield (start from scratch) and through package integration. The four approaches to creating or incorporating an SOA include top-down business architecture driven, bottom-up leveraging of legacy assets through componentization of externalized services, package integration and green field development.

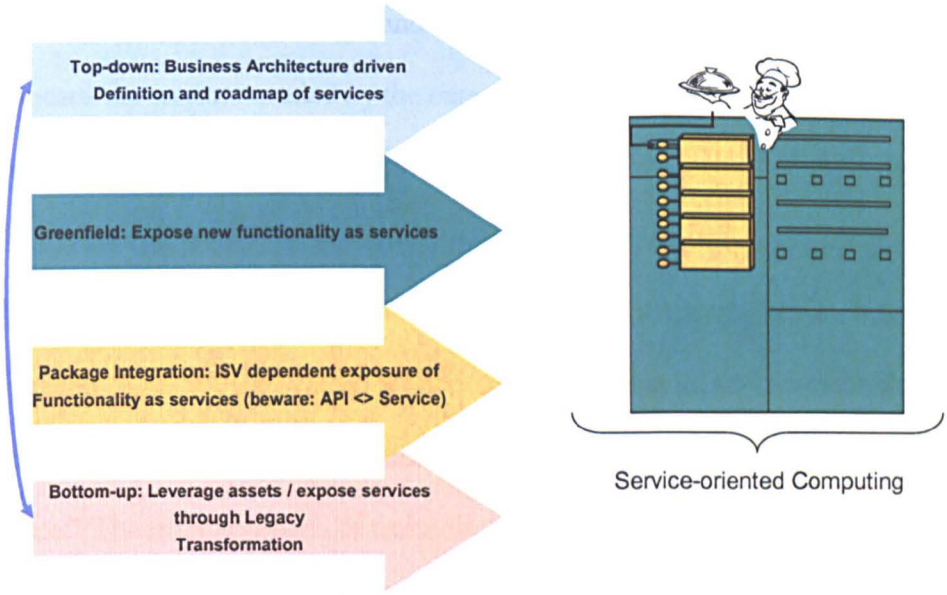


Figure 32: Four Approaches to SOA

The top-down business driven approach uses a decomposition of the domain into business processes, sub-processes and business use-cases. The business use-cases common to lines of business are usually done as a priority as they leverage reusable assets.

Services can also be created by leveraging existing functionality in legacy systems. The approach of legacy transformation with the process of legacy knowledge mining, externalization and componentization are generally used to achieve an SOA from this bottom-up approach.

Integration with or migrating to an SOA with packaged applications is tricky: packaged applications are often locked in terms of exposing their functionality in anything but the most rudimentary fashion through API's. The API approach is a fine-grained approach, differing from that of Services, which tend to be more coarse-grained and business aligned.

Greenfield refers to regular ground-up Java –based development that then exposes the services needed by the enterprise and determined by the top-down business driven approach.

To succeed in migrating to an SOA, a combination of approaches is often required. Also, migrating to an SOA cannot be done in an ad-hoc technology-focused fashion: "buy the tools and generate the WSDL and the SOA is in place." The latter approach of technology and tool focus is unrealistic, error-prone and extremely risky: tools cannot partition or discover the right granularity or functionality of services that are business aligned and that can be composed into usable orchestrations through standards such as BPEL[6].

[6] Business Process Execution Language.

7.3 A Method or Development Approach for SOA

Therefore, a method or development framework is required whereby services are discovered (top-down) and leveraged (bottom-up; from legacy and packaged applications). This section introduces such a method and describes the IBM SOA Method. This includes a combination of techniques and steps that describe how to employ top-down and bottom-up techniques to succeed in creating/migrating to an SOA.

This method consists of seven main steps. Although there are many more details in standard practice, we will focus only on the most salient features of the method through these steps.

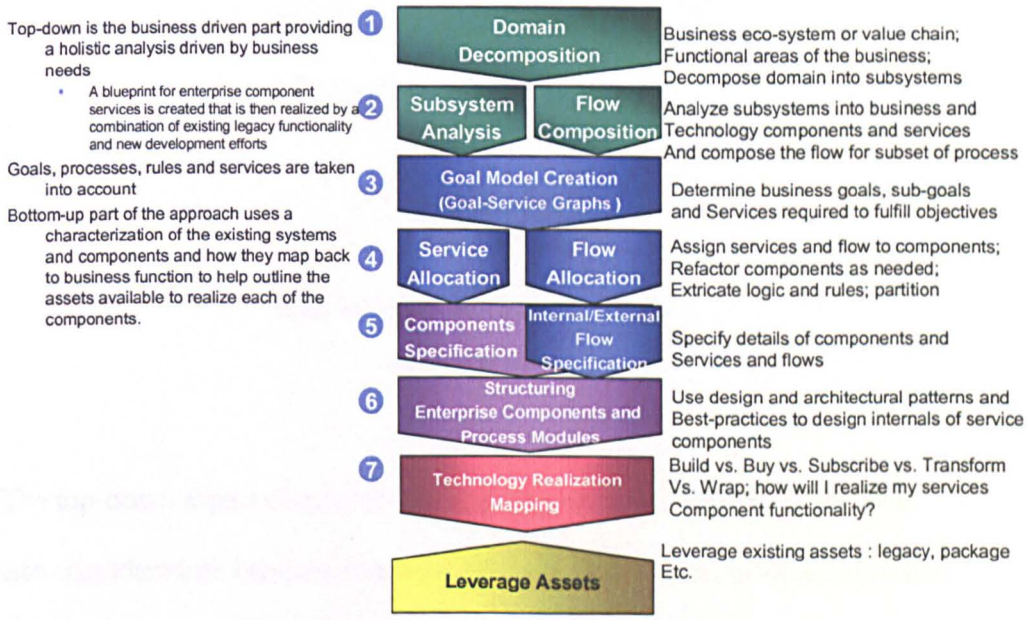


Figure 33: Seven Main Steps of SOA Development Framework/Method

Note that Figure 33 does not show the possible parallelism of activities, but simplifies the picture by assuming a more step by step approach. In practice, it is recommended to conduct some steps in parallel as depicted in Figure 34.

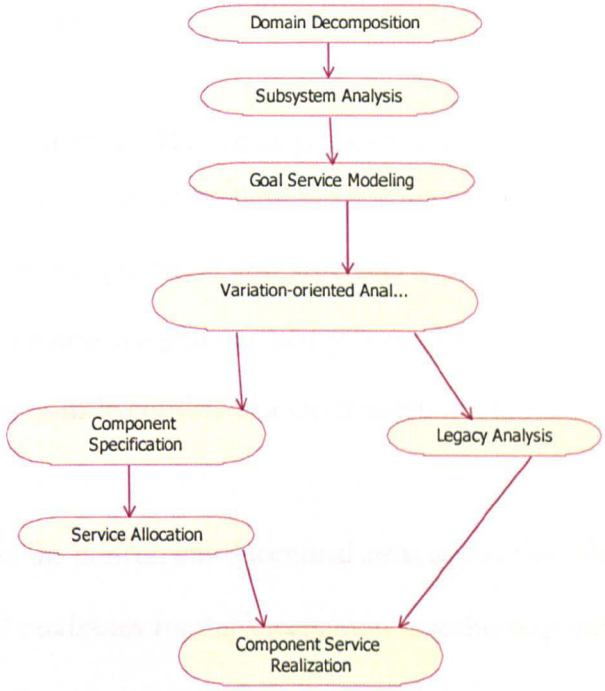


Figure 34: Some Parallel Steps in Executing the SOA Method

The top-down aspect comes from taking business perspectives and models into consideration: business function, process, sub-process, business use-cases are elaborated and form the outlines of component boundaries.

This creates a blueprint for a component domain model; allocating conceptual containers for which to categorize and later actually use to implement services

and chunks of business functionality often enunciated in the form of use-cases.

Components provide boundaries and containers for services (often discovered through use-case analysis and goal service model creation) to reside in.

Below we will outline the seven key steps necessary to support component-services architectures.

7.3.1 Domain Decomposition

In this step we analyze the domain into its constituent business architecture consisting of business processes, sub-processes and use-cases. Recall that every business process is a goal oriented process. From a business perspective, the domain consists of a set of functional areas.

We decompose the domain into functional areas across the value-net; these are often good candidates for implementation as technology subsystems.

Each Enterprise Component maps to a functional area or business process within the business domain.

- ***Enterprise Business Components are defined in a top-down fashion by taking into account the nature and evolution of the business, its goals, conceptual model, process model, rules policies models.***

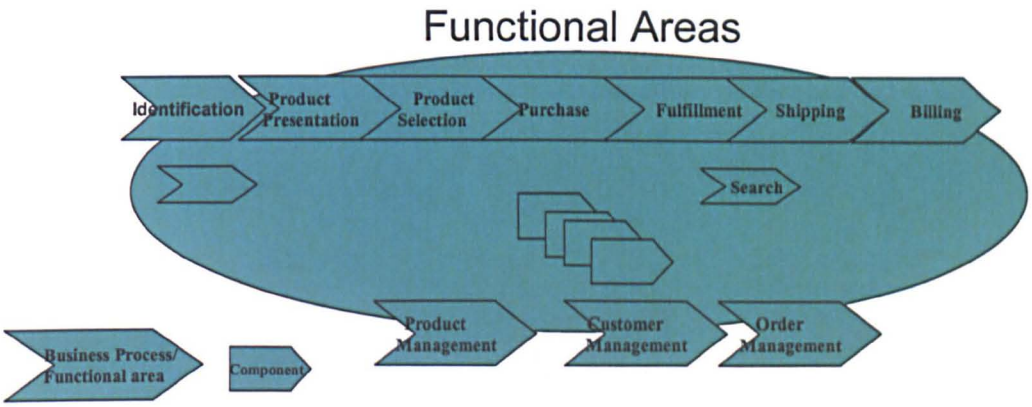


Figure 35: Step 1 Domain Decomposition into Functional Areas and Processes

In this step, we thus identify the functional areas across a value-chain or value-net and define the scope of the effort:

- Is it within the enterprise across one, two or more business lines?
- Is it across a value-chain within business partners in a supply chain?

Thus, the figure below depicts an overall business process

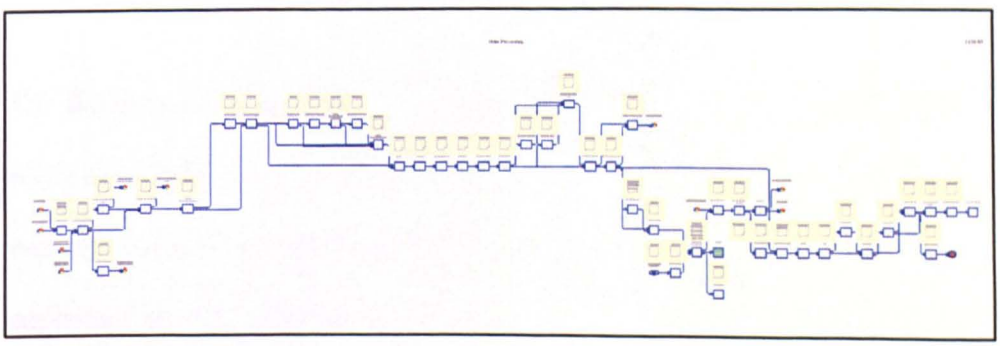


Figure 36 shows the decomposition of the domain into processes, sub-processes and business use-cases. These business use-cases are good candidates for business services, ultimately exposed as a WSDL interface (service description) on an owning enterprise component.

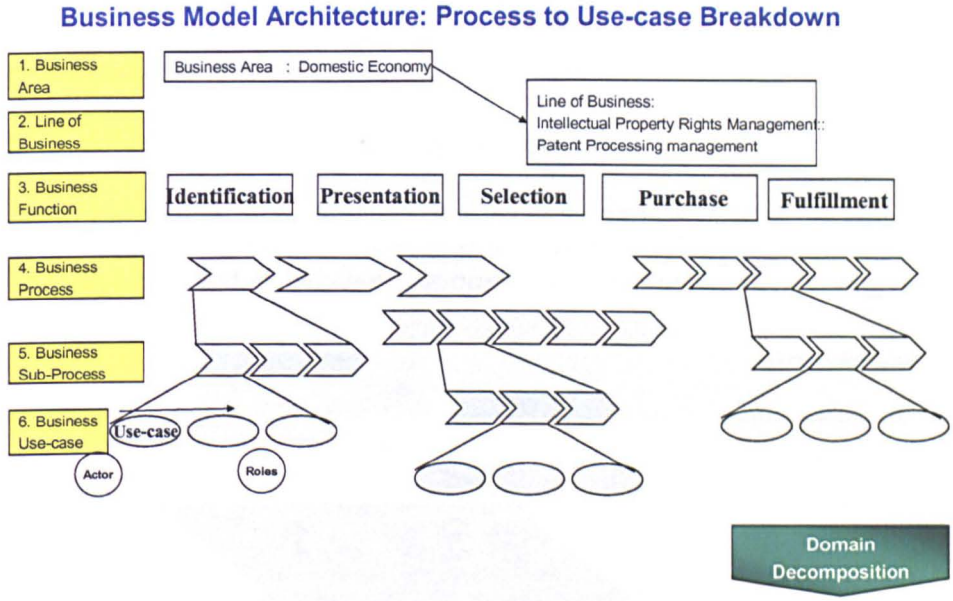


Figure 36: The Hierarchy of Domain Decomposition in Business Architecture Definition

These use-cases will now become the services that are exposed on the subsystem (later, enterprise component's interface as web services).

The Business use-cases thus serve as candidates for business services in the SOA; their definition was business driven and aligned, offering a common, reusable "chunk" of business functionality. In the business use-cases, it is important to note the data to be input and output from each service or

external invocation and have it defined at least at a high level, to be refined during component and service specification.

The term functional area is a business term. As we move into design, each functional area will be mapped to one or more subsystems in the architecture. Often the mapping is one-to-one as shown in Figure 37. So for each functional area we will often find at least one subsystem that will then be identified and elaborated as design continues.

Each Functional Area or Business Process can be thought of as an IT *subsystem that creates a natural business driven boundary for large-grained enterprise components that provide services.*

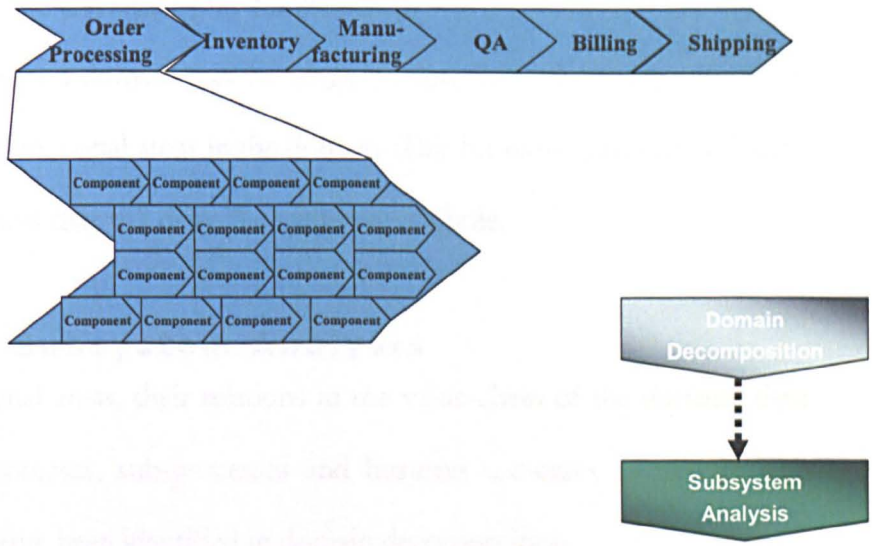


Figure 37: Functional Areas and Processes Map to Subsystems that will become Enterprise Components

Functional Areas are a business notion while the subsystems are a technology notion; there is a straightforward mapping between them; often one to one.

Object-oriented analysis and design (OOAD) tends to produce large object graphs. Hence, note the departure from traditional object-oriented analysis which would yield a large number of fine-grained objects first. Here, the larger encompassing (perhaps virtual) structure is first identified and them its constituent elements are refined (through a more top-down approach) or allocated (of we leverage legacy for example).

Next we create a domain specific language describing the composition and flow of the functional areas in the domain. This business grammar will serve as the manners that will drive the system as a whole.

7.3.2 Subsystem Analysis

The functional areas, their relations in the value-chain of the domain, their business processes, sub-processes and business use-cases as well as high level rules have been identified in domain decomposition.

Now, we are moving into more of design and into more of architecture and technology driven decisions. Subsystems are a large-grained unit of conceptual modeling that naturally follows a business driven top-down domain decomposition.

The business use-cases often collaborate to support a business process such as Figure 38. Subsystem Analysis refines the business use-cases into system

use-cases that support a given business process, often corresponding to the functional area's subsystem.

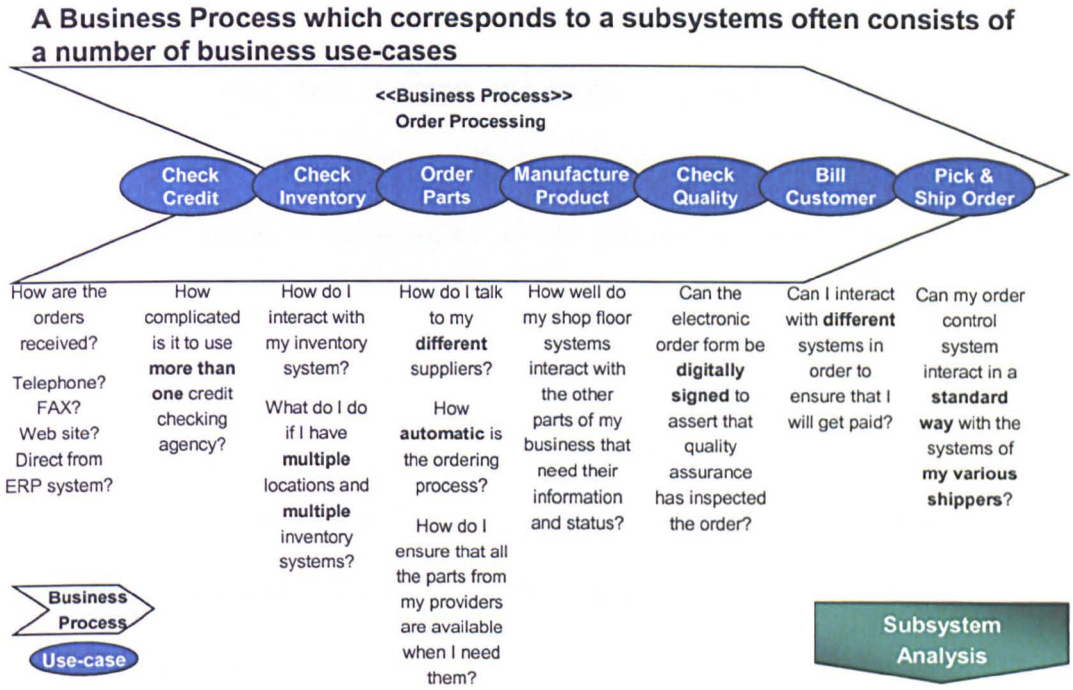


Figure 38: Use-cases collaborate to create a business process

During subsystem analysis, we take each subsystem identified in the previous step and decompose it into its set of constituent business and technical components. Thus, we delegate the functionality provided by the subsystem to fulfill the functional area's roles and responsibilities to a set of constituent business and technical components. Business components supply the business functionality required, while technical components provide the necessary technical infrastructure to needed to support the

business components and operate their functionality. Technical components are thus operational and non-functional oriented, while business components tend to satisfy functional requirements.

Thus, by composing these business and technical components, a larger-grained enterprise component can be constructed. This combination of business and technical components is not merely structural, but also involves the definition of flow: how do the business components collaborate to provide functionality to enact business processes? The functioning of the business components is enabled through the services (implemented by the component's finer level object or legacy system structure) defined and required by the business use-cases they support as shown in Figure 39.

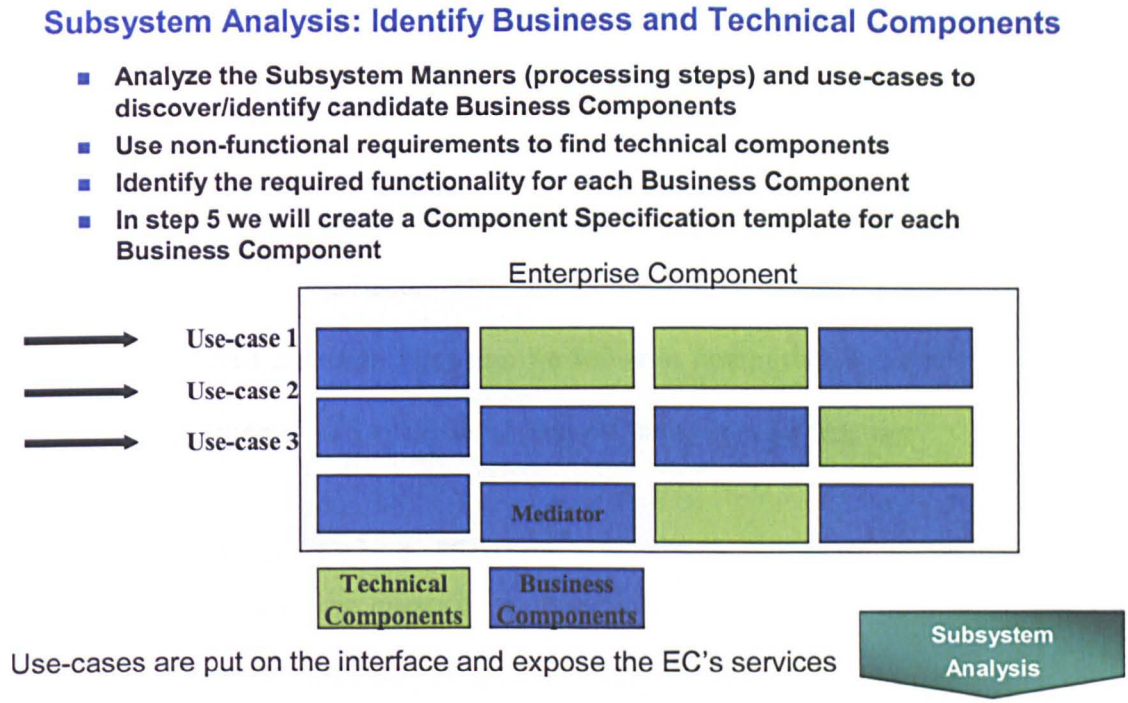


Figure 39: Step 2: Subsystem Analysis assigns use-cases as services and specifies finer grained components

The business use-cases will reside on the interface to these subsystems and the constituent business and technical components will supply the behavior necessary to support the service. This may require composite services; i.e. those built out of a combination of finer-grained services.

Hence in subsystem analysis, system use-cases, business and technical components, their dependencies and flow, the interfaces of the subsystem are determined.

A parallel activity is recommended which guarantees that the business services have all been identified. The next step, often done in parallel with subsystem analysis describes how new services that are traceable back to business requirements and goals are identified and the complete set of business services are thus defined.

It is key to note that for each subsystem we will then design the high level flow and composition of that subsystem using a domain-specific language.

7.3.3 Goal Service Model

Many times, in OOAD the question of how to arrive at Objects (object discovery or object identification) is raised as being a non-trivial, subjective activity. The underlining of nouns and intuiting names, events or roles are often cited. None of these activities is a precise heuristic which helps conceptualize the rationale behind the selection of an object in the domain. Similarly, service identification, which implies the discovery of business aligned services for the entire organization, presents a similar problem. Use-cases provide a necessary but insufficient condition for the identification of services in an SOA.

By interviewing business owners, querying them on the goals within the scope of work, we can create a tree of related sub-goals that are pre-requisites to the achievement of the initial high-level, often intangible and lofty goal. Each level of sub-goal is broken down to a set of further sub-goals until the services required to fulfill them are clear. This is called a goal-services model.

Create a goal-services model of the goals and sub-goals that must be realized in order support higher level goals. Associate the services required to support the realization of sub-goals. This will make services traceable back to the goals that the business indicates it needs to achieve.

A Goal Service Model (Example) defines sub-goals and services required to fulfill them.

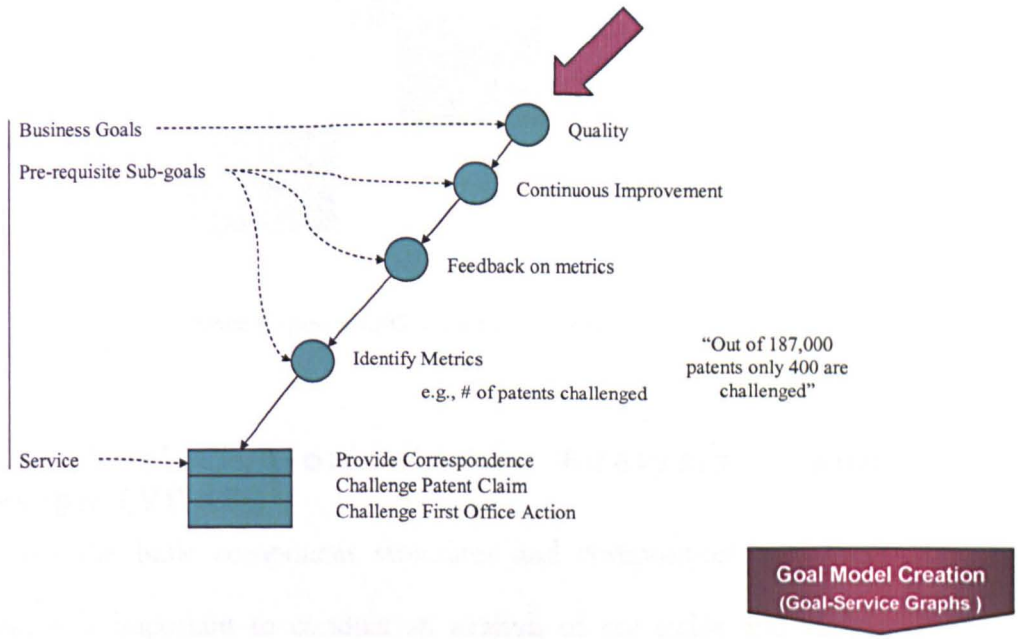


Figure 40: Step 3 : Goal-Service Model Creation

The generic form of the above diagram can be seen in the diagram below which has service associated with each level of goal/sub-goals.

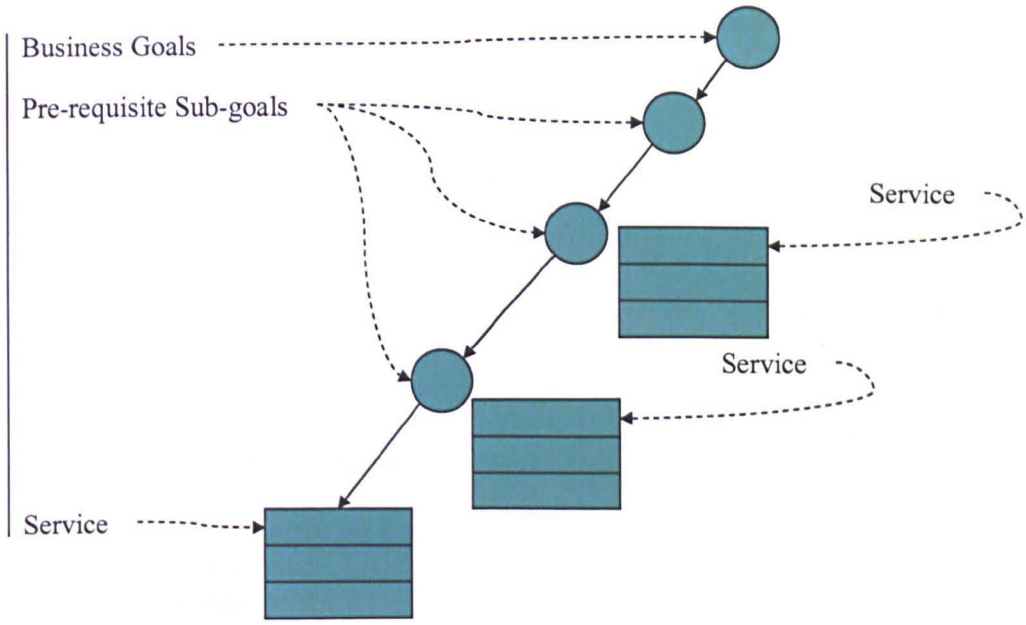


Figure 41: generic GSG (Goal-Service Graph)

7.3.4 Variation-oriented Analysis and Design (VOAD)

Now that the basic component structures and composition have been defined, it is important to conduct an analysis of the stable and variant aspects of those components in the context of the business domain and project that they will be used within. Apply the principles and techniques of VOAD to determine the extent of the externalizations of variations. For more details on VOAD and its seven principles see [10].

7.3.5 Business and Technical Component Specification

Now that the constituents of enterprise components have been identified, they need to be specified in detail. Elements such as those shown below can

be captured for each business or technical component that will participate in a release within the scope of the project.

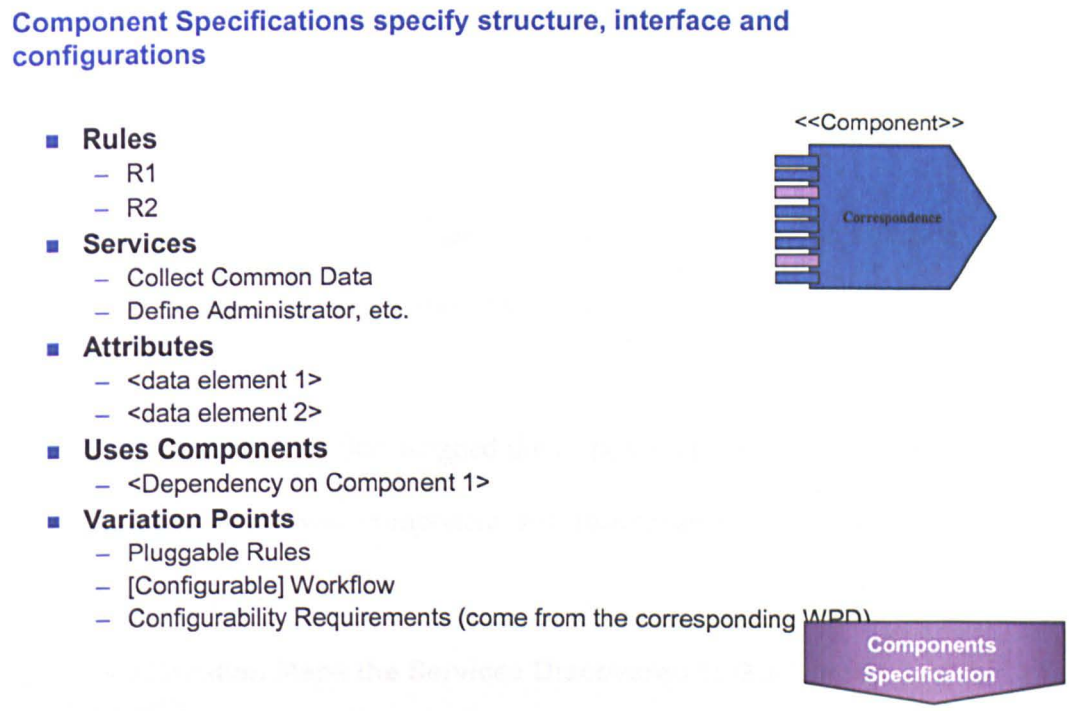


Figure 42: Step 5 : Component Services Specification

Thus, Components (business and technical) and their service descriptions are now specified with greater detail. The next section fills out the 'services" section of this template or adds to it to make sure it is complete.

7.3.6 Service Allocation

Now that services have been identified through a combination of domain decomposition and goal service modeling, we can create interface definition for the services (e.g., in WSDL). The next step is to find a home for them: who will provide the implementation? The answer will differ based on

whether you are a service provider or consumer. A service consumer will want to be able to flexibly replace its implementation based on non-functional characteristics (higher volume handled, less down time) and economic factors (cheaper service).

A service provider, on the other hand will want to implement the interface using one or more of its components or existing functionality (if not componentized).

Service allocation is a step that assigned the responsibility of implementation of the service to a given component for maintenance, governance and accountability aspects.

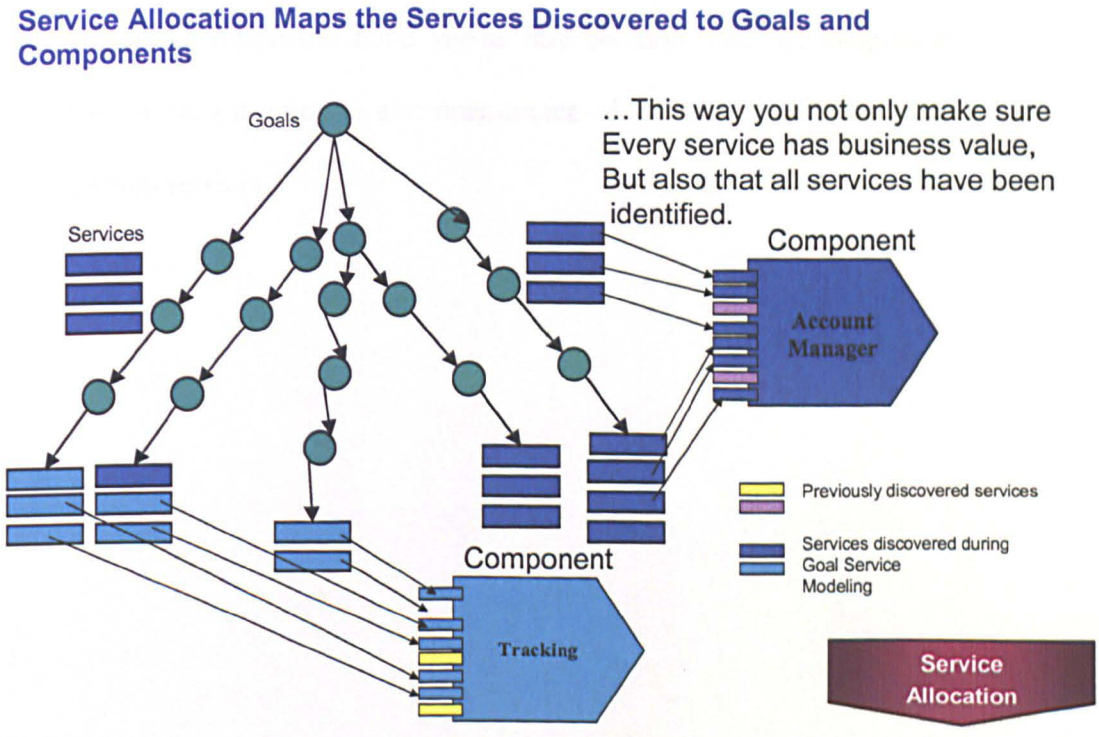


Figure 43: Step 4: Service Allocation

Once the goals and services have formed the list of necessary services, we take the components defined by domain decomposition, subsystem analysis and goal service model and assign them appropriate services.

7.3.6.1 Business and Technical Component Realization

Once the components have been specified in detail, their implementation mechanism and who implements them must be resolved. You can build everything from scratch. Or you can outsource it completely as a turn-key solution. Between these two extremes lies the most common needs of IT organizations: to decide what to build, what to buy. However, it is critical to note that these are not the only alternatives. There are various alternatives other than the traditional build versus buy decision; namely, integration, transformation, subscription and outsourcing of parts of the functionality, esp. via web services.

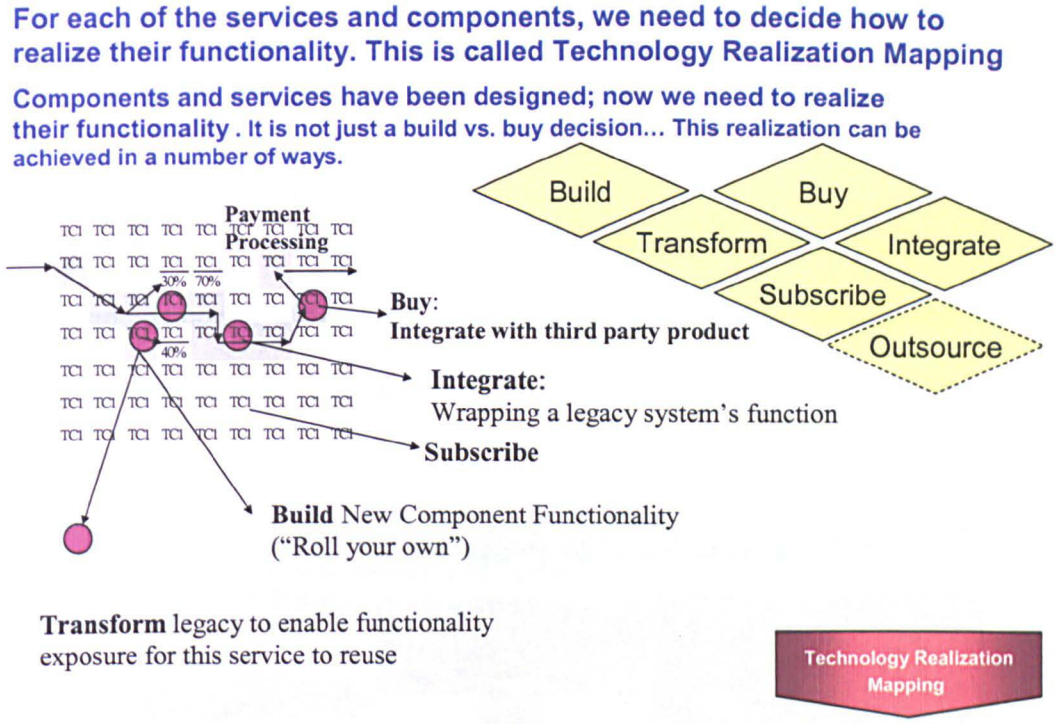


Figure 44: Step 7 Technology Realization Mapping

We decide how to realize the component by mapping it to a technology realization.

A table can be created such as in Figure 45: Technology realization mapping table for this purpose.

Business process	Sub-process	Use-case /service	Component	Current Implementation (if exists)	Future Technology Realization mechanism

Figure 45: Technology realization mapping table

7.3.7 Summary

To summarize, the method described here helps map a current business model and its underlying IT architecture to a service-oriented enterprise and application architecture as depicted in Figure 46: Current to Target SOA Architecture.

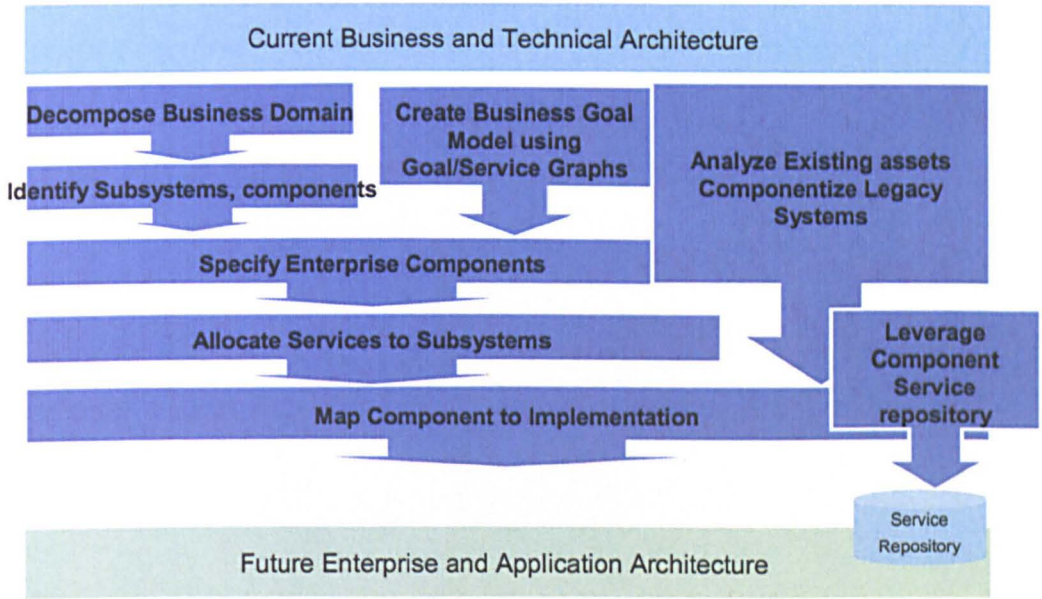


Figure 46: Current to Target SOA Architecture

7.3.8 Conclusion

That concludes the methodology extensions that grammar-oriented object design (GOOD) brings to the software engineering process.

The next section is the evaluation chapter and uses this method and applies it to an example called e-bazaar.

8 Chapter Eight: Evaluation and the E-bazaar Case Study

Objective

- To provide a case-study that demonstrates the implementation of GOOD using a realistic e-business project scenario

8.1 Mapping Business Architecture to Service-oriented Software Architecture

Grammar-oriented object design has been applied in industrial strength projects ranging from patent processing to telecommunications to financial services and higher education. Its related tools have also been used in that context. This case study is in reality taken from the experiences of three major projects that have been successfully executed in industry. The distillation of that experience has been cast in this canonical example called e-bazaar. E-bazaar explores an online order entry system similar to what would be found at a typical e-business website, for example, Amazon.com.

In this case study, we show how to start with high level business architecture and map it onto a service-oriented architecture using GOOD.

In order to accomplish this we have extended current object-oriented methods to include full spectrum (from high level analysis down to implementation) support for component-based and service-oriented software engineering. This is essential to ensure a consistent and repeatable way of analyzing and designing software systems with the GOOD approach [14]. These extensions are discussed in the next section. In the next section we show how we have extended current methods such as the Unified Process of Software Development with additional activities and artifacts such as Subsystem Analysis in which the Business Domain is partitioned into

cohesive business-process level subsystems with appropriate "manners" allocated to each subsystem using a *use-case grammar.*

8.2 Extending the Unified Process for Business Component-Based Development

We contend that the Unified Process of Software Development can be extended, and thus used for a Component-based Development methodology. Figure 1 illustrates the Unified Process (UP) with the following extensions: Architectural analysis (1), Variation-oriented Analysis (2), Architectural design (2a), Variation-oriented Design (3) and Subsystem Design (3a) and Grammar-oriented Object Design (4). Of these, Architectural Analysis is lightly impacted and Architectural design and Subsystem Design are lightly altered.

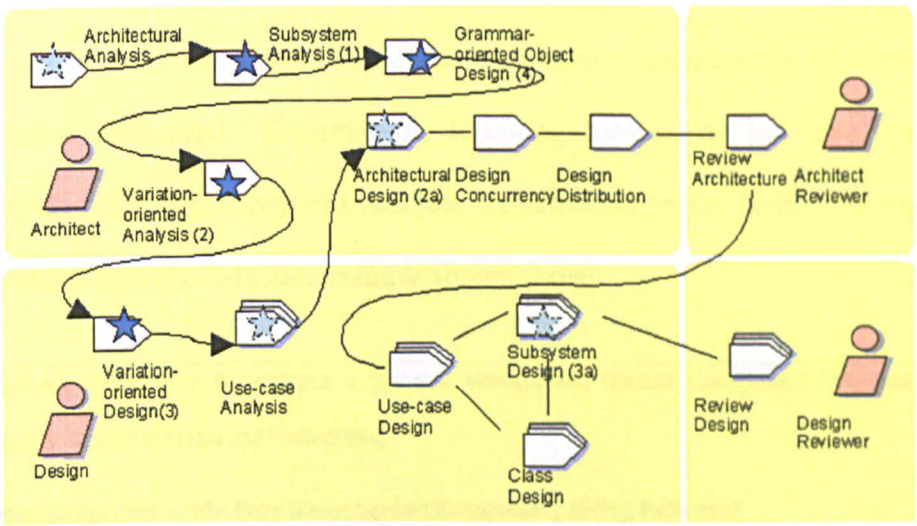


Figure 47: A Workflow Map for extensions to the Unified Process of Software Development.

8.2.1 Subsystem Analysis Example

An experienced architect uses patterns that are distillations of past experience that are known to work and provide effective solutions to clients across multiple projects [42]. The introduction of architectural mechanisms occurs here in the UP workflow. Consider how the world would appear to an experienced software architect: instead of a large number of fine-grained objects with little reuse potential, the problem domain would be partitioned into a set of subsystems that could repeatedly be reused in product lines and families of applications across multiple projects in various business lines.

In every "incarnation", i.e., instantiation, each subsystem might be implemented in a slightly different way that depends on the component realization in the target problem domain context. However, the set of types, i.e., interfaces, and manners that define the cluster comprising the subsystem remains unchanged. Consider the following subsystems that have been identified through Subsystem Analysis, an extension to the Unified Process workflow, for the E-Bazaar example shown above:

Customer Relationship Management = {Account Management, Contact Management (Addresses), Security, Customer Profile and Preferences}

Order Management={Order Entry {Shopping Cart Management}, Billing, Fulfillment}

Product Management = {Products, Catalogs, Pricing}

Inventory Management = {Fulfillment {Picking and Shipping}, Vendor Management}

Financial Management ={Billing, Accounting and Bookkeeping, Accounts Payable, Accounts Receivable}

Note that Shopping Cart Management can be thought of as part of the Order Management subsystem. However, this is architecturally significant so that in Subsystem Analysis we consider it to be a separate subsystem abstraction. The use-case grammar showing their relationship with regard to the larger-grained enterprise-level subsystem we are calling Online Purchase Subsystem can be written as:

> *Online Purchase Subsystem = {Customer Management, Order Management {{Shopping Cart Management (Order-Entry)}, Order-Processing, Fulfillment, Billing}, Product management}*

The above domain grammar tells us that in order to build reusable components for a larger Online Purchase Component we must build at least four other components relating to the handling of Customers, Products, Shopping Carts and Orders. Orders pertain to the back-office activities that occur once an online purchase has been made. The Shopping Cart metaphor is used to set up the order. Once the order has been confirmed, an Order is created that is sent to Accounting (to generate an invoice or shipping list), to Inventory in order to Pick and Ship the product. Inventory sends this on to Shipping, which actually does the physical shipment.

Let us consider a use-case within the context of an online order management system we call *E-Bazaar*. Especially noteworthy is the representation of a business domain-specific language in the form of use-case grammars included in each use-case description.

8.3 E-BAZAAR USE-CASE: MAKE AN ONLINE PURCHASE

8.3.1 Overview

This use case describes the process of making an Online Purchase using the E-Bazaar System, a hypothetical online e-business system that presents catalogs of items for customers to order over the World Wide Web.

Actors. Customer (Online Customer via Web access channel), Shipping Vendor, Product Catalog, Credit Verification System, Address Sanitizer.

8.3.2 Flow of Events: Basic Flow

1. This use case starts when the actor initiates a Purchase for an Online Product in the E-Bazaar System after having browsed and selected items for purchase.

2. The system displays a product item that the user had selected. The user adds the selected product item to their shopping cart. This process (selecting and adding an item to the shopping cart) is repeated until the user is satisfied with the items in his shopping cart. In order to initiate the online purchase, the user selects Checkout.

3. The system displays an Order Summary (an itemized list of selected items and their subtotal). The user reviews the items and clicks "Continue".

4. The system brings a billing and shipping address screen with the user's default billing and shipping information. The user either selects the default or selects an alternative shipping or billing address and clicks continue.

5. The system displays the default payment method with the last four digits of the credit card number for user verification. The user accepts the payment method or selects an alternative payment method and selects "continue".

6. The system displays an itemized Order. This includes items, subtotals, taxes, shipping charges and total. The user then reviews the Terms of Agreement and checks the "I agree" check box and clicks on "Submit Order".

7. The system submits the order to the E-Bazaar and displays a confirmation of the purchase to the user, along with a thank you note and a confirmation number.

We will now write a grammar for the above basic flow scenario. In the early phases of analysis, when doing high level use-case modeling, a list of use-cases and actors is produced. This list of use-cases, often depicted in a use-case diagram presents no information about the sequencing of the use-cases or the scenarios within them. Thus there is a gap between the high level business process description, the use-cases that participate in fulfilling the objectives of the business process and the algorithmic representation needed to show the

flow of events, services and components that are involved in the realization of the use-case.

Often a sequence diagram is used to realize a use-case; while the more global interaction between use-cases is not specified.

Representation of the use-case or the flow of a set of use-cases (not shown here) through a use-case grammar bridges the gap between the business description and an I/T specification by creating a business level specification through GOOD that can be executed.

We have developed a Tool called the Business Compiler that executes the specification provided by a use-case grammar. A use-case grammar is a representation of the manners of the use-case or system.

8.3.3 Use-Case Grammar (Manners Implementation)

A use-case grammar is the manners written to describe the domain specific behavior of a set of related and interacting use-cases. The use-case grammar shown here describes the higher level flow of the e-bazaar application.

```
Online Purchase = {Identification, Presentation, Selection, Purchase, Confirmation, Order Fulfillment}

Identification = {Challenge User with Login, Verify UserID and Password}

Challenge User with Login = {}

Verify UserID and Password ={}
```

```
Presentation = {Display Menu}

Display Menu ={}

Selection = {Browse Product Catalog, Select a Product Item, Shopping Cart Operation, Checkout , Selection1}

Browse Product Catalog ={# pol}

Select a Product Item ={}

Shopping Cart Operation = {

        {Add Item to Shopping Cart | Delete Item From Shopping Cart | Save Shopping Cart },
RepeatShopping1}

RepeatShopping1 = {{stop, Epilogue}|{continue, Shopping Cart Operation}}

Selection1 ={{stop, Epilogue}|{continue, Selection}}

Add Item to Shopping Cart ={add Item to Shopping Cart}

Delete Item From Shopping Cart ={delete Item From Shopping Cart}

Save Shopping Cart = {save Shopping Cart }

Checkout = {Complete Order Info}

 Complete Order Info = {{Verify Billing and Shipping Address| Select Billing and Shipping Addresses},

        {Verify Shipping Method | Select Shipping Method}}

Verify Billing and Shipping Address ={verify Billing and Shipping Address }

Select Billing and Shipping Addresses = {select Billing and Shipping Addresses}

Verify Shipping Method ={verify Shipping Method}

Select Shipping Method ={select Shipping Method}

Purchase = {Review Order, Review Terms of Agreement , {{Submit Order} | Change Order to Quote}}

Review Order = {{review and accept, Acknowledge Terms of Agreement} | {reject, Epilog}}

Review Terms of Agreement = {}

Acknowledge Terms of Agreement = {}

Submit Order ={submit Order | Cancel Order}

Cancel Order ={cancelOrder}
```

Change Order to Quote ={change Order to Quote}

Confirmation = {Send confirmation number to user}

Send confirmation number to user ={send conf | not send conf}

Order Fulfillment = {Pick and Ship Order}

Pick and Ship Order = {# msg(pick), # msg(ship)}

Epilogue ={# msg(Done)}

Key: Non-terminals start with an uppercase letter; terminal symbols with a lowercase and actions or service requests are prepended by a "# ".

The use-case grammar is a new artifact that consists of a domain specific language specifying the manners of the systems. It combines the notion of a structured use-case with one of subsystem partitioning and domain-specific languages. Once a domain analysis [4] is conducted and business language analysis [69] is completed, the key abstractions of the domain are partitioned in terms of interacting subsystems that may eventually be realized as software enterprise components . Manners are assigned to each subsystem based on the Business Rules governing its behavior [88]. Subsequent Variation-oriented Analysis is conducted to separate the changing from the more stables, less-changing aspects and features; verify what changes; handle changing aspects using patterns [42][39]; partition the domain into subsystems and define manners for each subsystem and their interactions; and, use three layers of interface, abstraction and concrete realization in the aggregate inheritance pattern.

8.3.4 Business Grammars

The set of all use-case grammars within a business domain is called a *business grammar*. A business grammar can be written for each industrial domain such as Telecommunications, Banking, Higher Education, Healthcare, Distribution and Manufacturing. Such a set of Business Grammars can be used jumpstart projects and create a revolution in the software manufacturing industry. A given domain's generic business grammar (GBG) can be customized to meet the needs of a given project. The project team does not have to "start from scratch" and can use established business knowledge of the domain to serve as a starting point for software development projects. A company having such assets will have a significant competitive advantage in the marketplace.

The main driver for adoption is not only ubiquity of XML standards for business process execution but also the availability of editing and execution tools. The first of these tools have been developed as the Business Compiler.

Business Grammars generate Business Languages.. A Business Domain Specific Language (BDSL) is an industry or business domain-specific language that characterizes the key manners or rules governing the behavior in the domain's partitioned set of subsystems. There have been numerous examples of highly successful implementations of software based on Domain-Specific Languages (DSLs) [1, 4-7,53,76], none of which have used a DSL in this capacity.

Grammar-oriented Object Design (GOOD) uses DSLs in a specific manner. GOOD is concerned with identifying the Business Language for a given business domain, partitioning the domain into subsystems based on Subsystem Analysis, identifying variations within the subsystem manners and applying necessary patterns through Variation-oriented Analysis and Design, writing use-case grammars that define the Manners for the subsystem and the context in which it will interact with other component interfaces, once deployed and executing the control flow in a component framework through pluggable micro-workflows that implement the manners.

8.3.5 Problem Statement

The e-bazaar is a generic customer to business e-business venue where products are categorized into catalogs, they are browed, selected and added to a shopping cart metaphor for purchase. An order is submitted, a payment is made and shipping information supplied. Whereupon the business (supplier, service provider) will procure, produce, manufacture or pick and ship the ordered products to the client after obtaining the necessary monetary transaction.

8.3.6 Architecture Analysis

8.3.6.1 Domain synopsis

Customers maintain Accounts. Accounts contain Security information, Address information and preferences.

Customers use a Shopping Cart metaphor to Add items they have selected as a result of browsing the Product Catalog. The Catalogs and Products are to be found on legacy databases.

Once they decide to checkout, a Customer submits an order after reviewing the details of the order and confirming the shipping, billing addresses, method of payment and shipping method.

Note: When the architect creates a set of key abstractions, s/he does so based on repeated experience with the domain or with similar domains. Typically,

the architect will have had patterns of interactions between key abstractions from other projects that s/he comes to bear on the current one. Thus, although there may be many key abstractions based on patterns in the business domain, they are/can be grouped in terms of their reuse-levels at the subsystem level. At this level, a subsystem encapsulates the complexity it holds by exposing only one subsystem Façade that may include other more detailed abstractions within it. Remember that since we are targeting a higher level of reuse than merely the class, we may have a subsystem as the key abstraction (clusters of collaborating classes with a façade around them such as Order, order line item, product and price).

The following table summarizes the key abstractions with their corresponding descriptions:

Summary of Key Abstractions

- Customer
- Shopping Cart
- Product
- Catalog
- Order

8.3.6.2 Internal System Abstractions

The internal abstraction of the e-bazaar system can be found in the following table.

Candidate Subsystems	Abstractions within Candidate Subsystem	Description
Customer		Customers are online shoppers who've made a purchase at e-Bazaar
	Address	This is a generalization of mailing address and shipping address that the customer records as part of the maintain customer accounts use case. Customers can hold multiple addresses for both shipping and billing.
	Account	The customer creates an account, which includes their shipping address, billing address, preferred shipping method, method of payment and secure login information (user ID and password). An account contains a customer's preferences: default catalog, preferred item types, and other default behavior that the customer would like to customize in order to have a personalized Web experience.
Shopping cart		A Shopping Cart is a metaphor used in online shopping to provide the customer with a container and an intuitive way of adding items to it, as they browse products and select the ones they want to purchase. The shopping cart can have items added to it and deleted from it. A shopping cart can also be saved for later reference. The customer can always request to view the contents of their shopping cart.
	Line Item	A Line Item in a Shopping cart.
Order		The Shopping Cart is a temporary construct designed to capture the references to Products (aka items) until the Customer submits an order. At the backend, an Order is now created and persisted for audit purposes. The Order is tracked and will be

		picked and shipped and linked to the accounting System.
	Order Line Item	A Line Item in an Order.
Product		The product is an offering that the seller provides to a buyer for some exchange of funds. Products are organized according to type, and stored within product catalogs. A product has a price associated with it. A products' attributes and availability can be determined by asking the cart.
Catalog		A catalog is a named grouping of product items that are available as service offerings for buyers to purchase. The product management department of e-Bazaar maintains catalogs and products. The Product Management domain deals with defining new catalogs and putting products into catalogs.

8.3.6.3 External System Abstractions

The external system abstraction can be found in the following table.

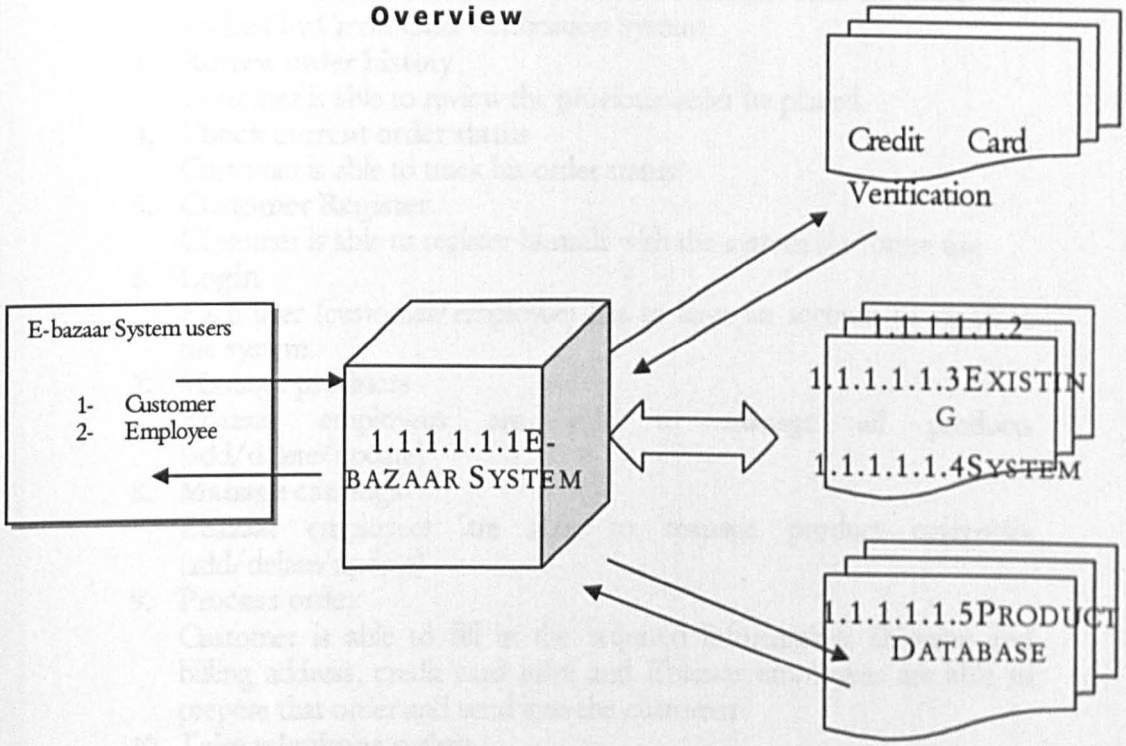
Candidate Subsystems	Abstractions within Candidate Subsystem	Description
Product catalog system		This is a legacy system, which contains the description of all the products that e-Bazaar markets to date. It also includes a description of the groupings of those products in the form of a catalog.
Credit verification system		The credit verification system is an external entity that is responsible for verifying the credit worthiness of the potential online customer who has entered a method of payment as part of their customer account, and is now attempting to submit an order.
Address Sanitizer		An external system that given an address in the USA determines if this address exists or is a valid US address.
Shipping Company		The shipping company is the independent vendor who is responsible for shipping the

		items.

8.3.7 Domain Decomposition

The following is the domain decomposition for the e-bazaar case study.

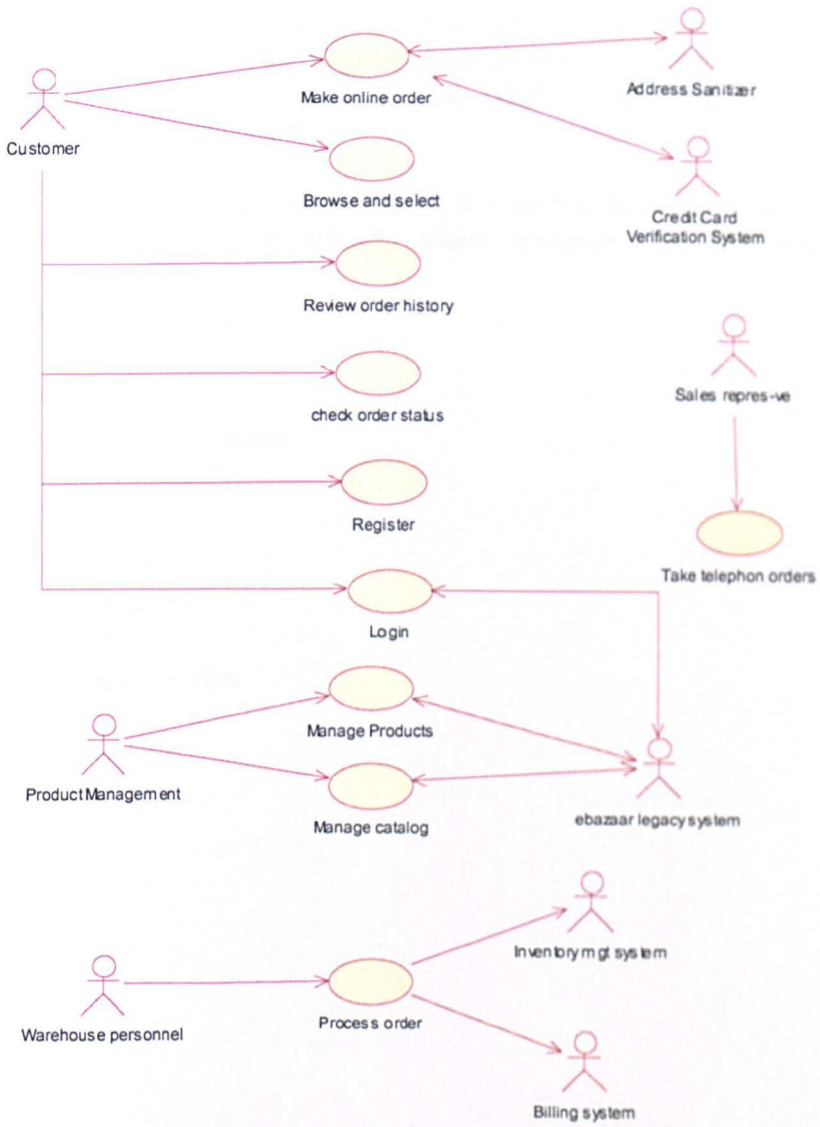
8.3.7.1 E-bazaar Context Diagram and System Overview



8.3.7.2 Use-cases

1. **Browse online catalog**
   Customer is able to browse the main catalog, select products and add them to the shopping cart
2. **Make online order**
   Customer is able to make online orders after filling the shopping cart, customer address sanitized by Address Sanitizer. And his credit card verified by Credit Card Verification System
3. **Review order history**
   Customer is able to review the previous order he placed.
4. **Check current order status**
   Customer is able to track his order status
5. **Customer Register**
   Customer is able to register himself with the system for future use
6. **Login**
   Each user (customer/employee) has to have an account to logon to the system
7. **Manage products**
   Ebazaar employees are able to manage all products (add/delete/update)
8. **Manage catalogs**
   Ebazaar employees are able to manage product categories (add/delete/update)
9. **Process order**
   Customer is able to fill in the required information; shipping and billing address, credit card info; and Ebazaar employees are able to prepare that order and send it to the customer
10. **Take telephone orders**
    Ebazaar employees still using the legacy way of purchasing, which is by phone.

Use-Case Diagram



8.3.7.3 Business Functionality

This section provides some direction on the relative importance of the proposed system features. As development progresses in the system, the features attributes will be used to weight the relative importance of the features and plan the release content.

It is anticipated that E-bazaar system will be release for general use at E-bazaar through the following release, which contains the following functionalities

- Browse online catalog
- Make online order
- Review order history
- Check current order status
- Customer Register
- Login
- Manage products
- Manage categories
- Process order
- Take telephone orders

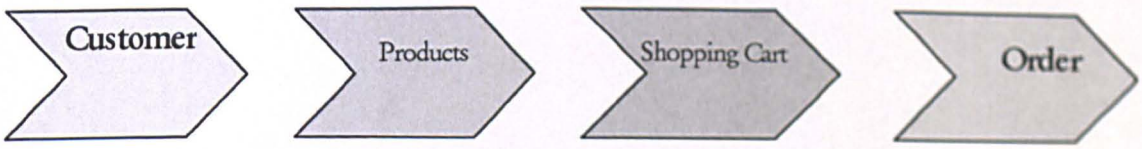
8.3.8 Subsystem Analysis

8.3.8.1 Selected subsystems

According to Ebazaar system analysis, the following subsystem with their components will be used

- Product Catalog
  - Catalog Subsystem Facade
  - Catalog
  - Product
  - Mediator
- Shopping Cart
  - Shopping Cart Facade
  - DBShopping Cart
  - DBShopping Cart Transactions
  - Shopping Cart Controller
  - Shopping Cart DBConnection
- Customer
  - Customer Subsystem Façade
  - DBCustomer Record
  - Customer
  - DBLogin
  - Login
- Order
  - Order Subsystem Façade
  - DBOrder
  - Order
  - Order Items
  - OrderSC

Subsystems process flow



8.3.8.2 Subsystems relationships

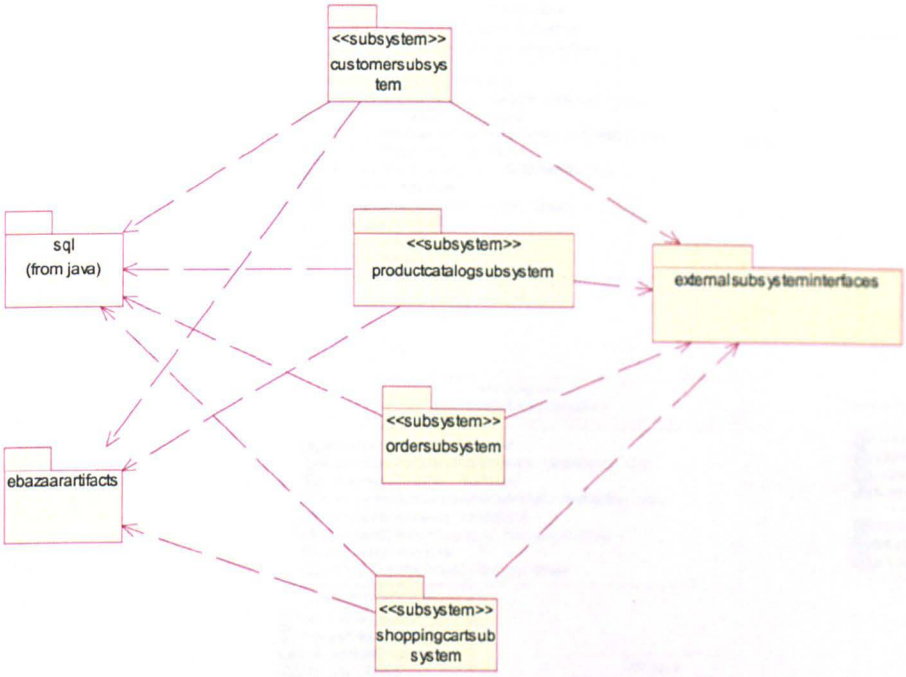


Figure 48: Component Dependencies

8.3.8.3 Customer subsystem

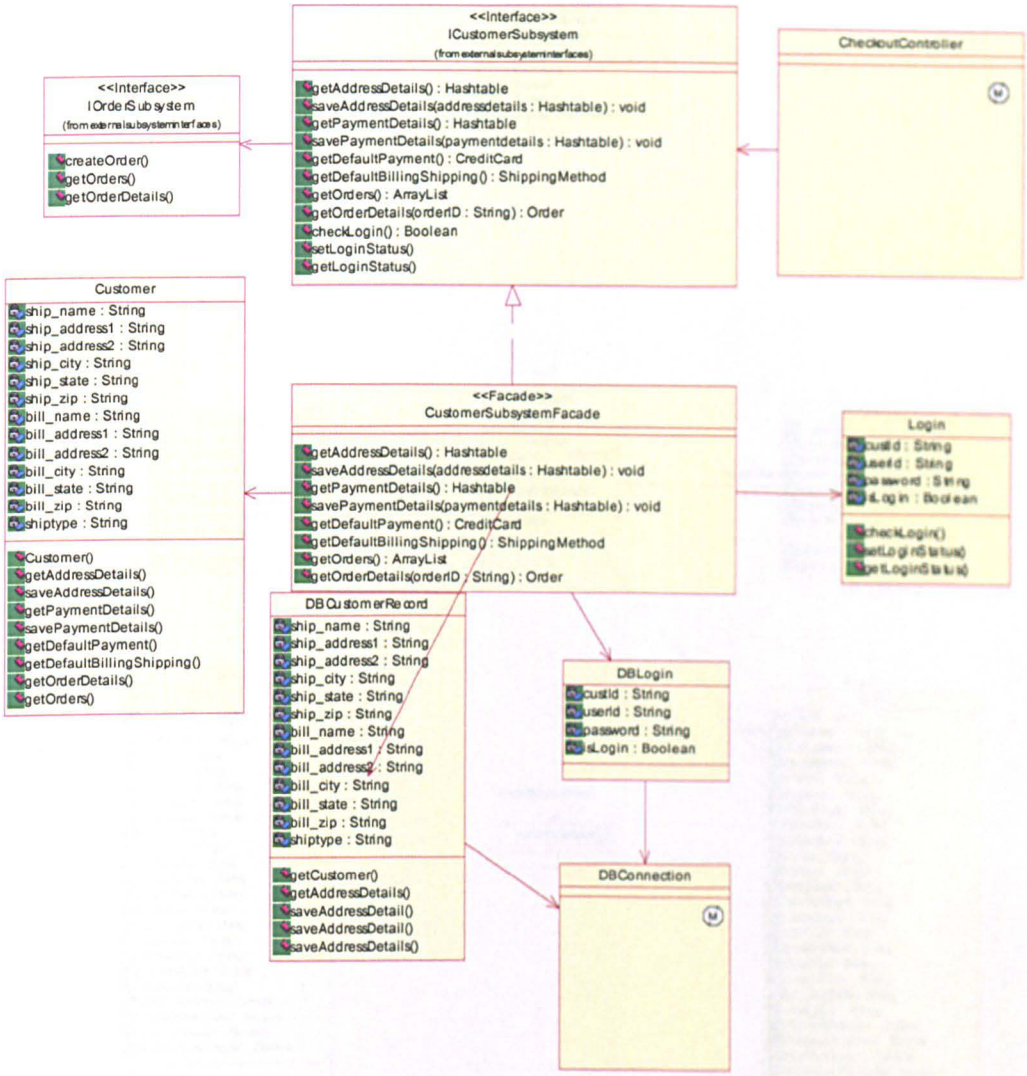


Figure 49: Customer Subsystem Enterprise Component

8.3.8.4 Order subsystem

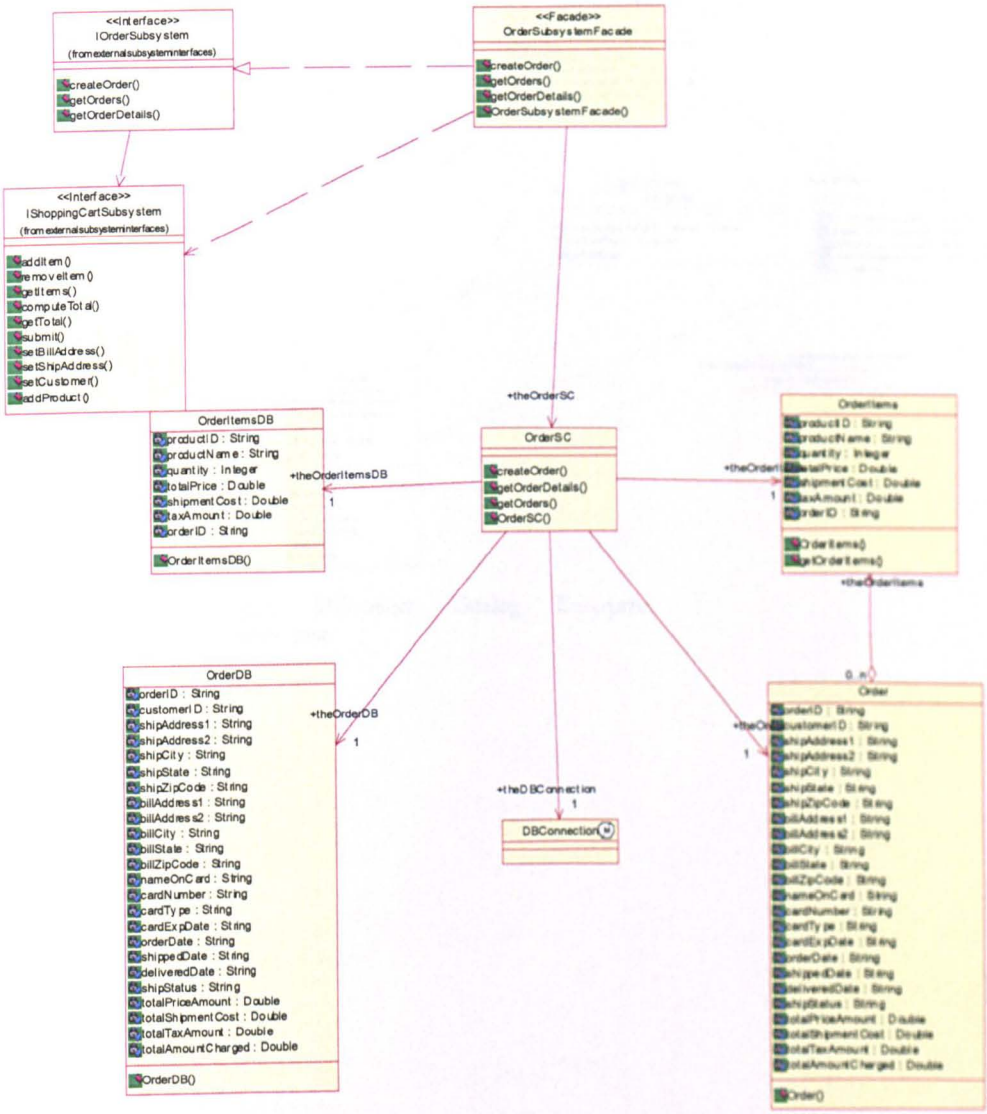


Figure 50:Order Manager Enterprise Component

8.3.8.5 Product Catalog subsystem

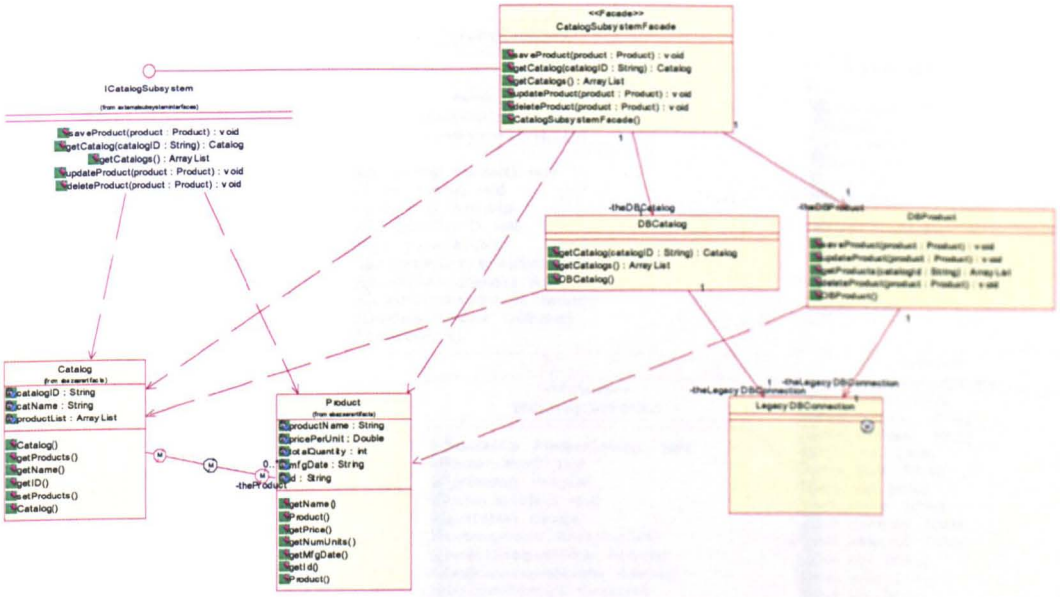


Figure 51:Product Catalog Enterprise Component

8.3.8.6 Shopping Cart subsystem

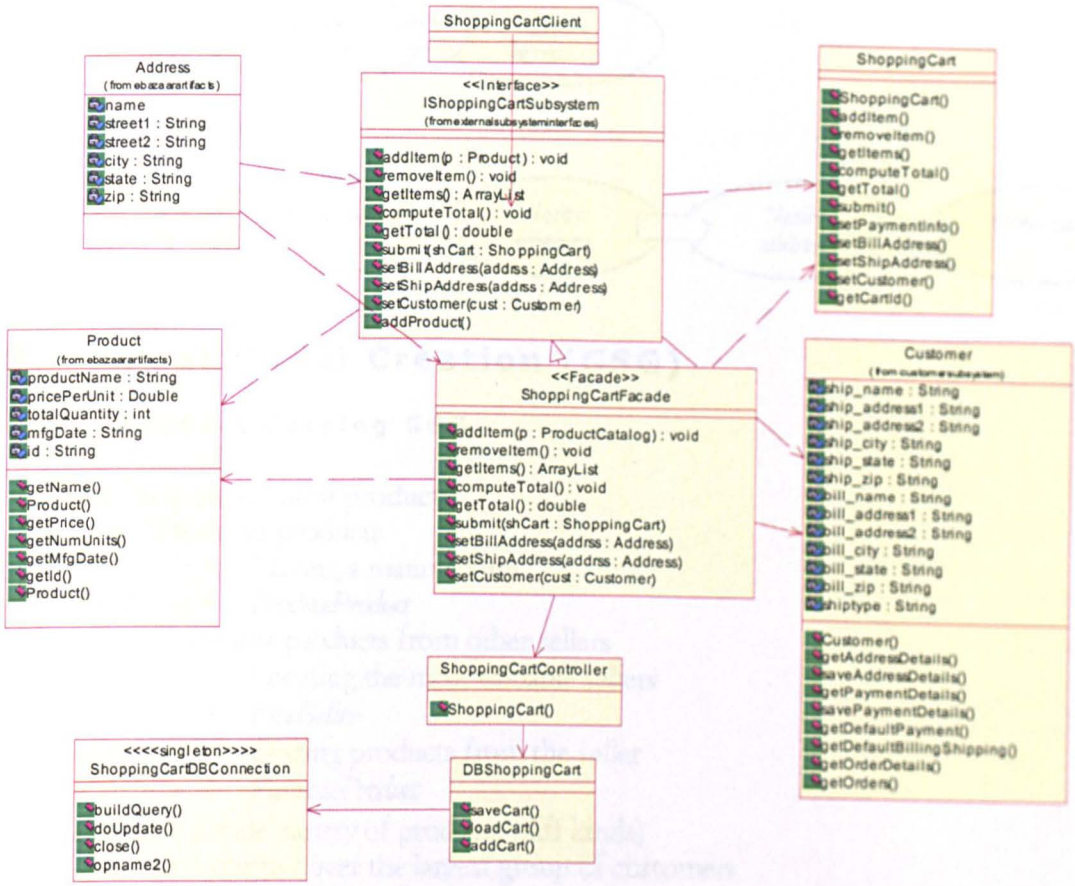
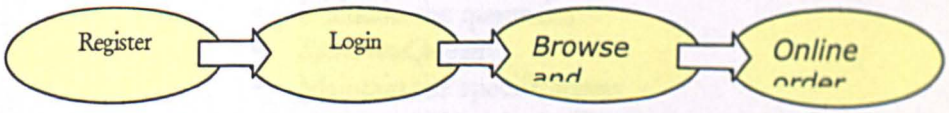


Figure 52: Shopping Cart Enterprise Component

8.3.8.7 Subsystem business processes

Customer processing



Order processing

Product catalog processing

Shopping cart processing

8.3.9 Goal Model Creation (GSG)

8.3.9.1 Product Catalog GSG

- Must have all the latest products
  - Produces products
    - Having a manufacture
    - *ProduceProduct*
  - Purchases products from other sellers
    - Locating the most valuable sellers
    - *FindSeller*
    - Getting products from the seller
    - *PurchaseProduct*
- Must have a wide variety of products (diff kinds)
  - In order to cover the largest group of customers
    - Increase chances of buying per day
      - Marketing
        - Awareness
        - Special offers
        - Cross-sell from other web sites or partners
- Must have the products grouped into catalogs for easy access
  - Maintain the product catalogs from time to time
    - Maintain the quantities
    - *StoreNewQuantity*
    - Maintain the specifications
      - *AddCatalog/AddProduct*
      - *DeleteCatalog/DeleteProduct*
      - UpdateCatalog/*UpdateProduct*
      - Move them to temporary store
      - *TransferCatalog/TransferProduct*

8.3.9.2 Shopping Cart GSG

- Easy to Use
  - User-friendly Interface
- Must have features clients need
  - Research competitors for features (➔) <this is a manual process >
  - Browse Catalogs
  - *ShowCatalog*
  - Select product
  - *ShowProduct*
  - Show Product detail
  - *ShowProductDetail*
  - Add to shopping cart
  - *AddProductToShoppingCart*
  - Delete Item
  - *DeleteItem*
  - Update Quantities
  - *UpdateQuantities*
  - Update Shipping Method
  - *UpdateShippingMethod*
  - Save Shopping Cart
  - *SaveShoppingCart*
  - Continue shopping
  - Proceed to checkout
  - Existing customer, just login using his account
  - *GetCustomerLogin*
  - New customer, has to create an account, and will be automatically logged on.
  - *GetCustomerLogin*
  - Show billing and shipping addresses
  - *ShowBillingAddress*
  - *ShowShippingAddress*
  - Change billing and shipping addresses
  - *SaveBillingAddress*
  - *SaveShippingAddress*
  - Ship to the selected address
  - Payment processing
  - *ShowPaymentData (if it exists)*
  - *SavePaymentData (in case of changes)*
  - Submit the order (*The order will be saved for tracking purpose*)
  - *SaveOrder*
  - Show order receipt
  - *ShowOrderTransaction*
  - Wish List

    - *ShowPreferredItems*
    - My Account
    - *ShowCustomerData*
    - View Cart
    - *ShowShoppingCart*
    - One-click Shopping
        - Get customer billing, shipping and payment info
        - *SubmitOrder* (submit the order in case billing, shipping, and payment data is set for each customer)
    - Price Comparison
        - *ComparePrice*
- Having a Pleasant Shopping Experience
    - Returning customer
        - Keeping track of customer's profile
            - Provide the means for customer registration
                - *Register Customer*
                - Keeping track of the history of purchases for each customer
                - *Save Purchase History*
        - Maintain number of purchases for each customer
        - *UpdatePurchase*
- Provide Capability for making Cross-sells
    - Show Customer what other purchasers have bought
    - *ShowOtherPurchased*
    - Show customer favorite items may come with the selected product
    - *ShowFavoriteItems*
- Rapid Customer Fulfillment and Satisfaction
    - Easy access to all catalogs
        - Listing all catalogs
        - *ShowCatalogs*
            - Listing products for each catalog
            - *ShowProducts*
                - Listing product detail
                - *ShowProductDetail*
                - Adding product to shopping cart
                - *AddItemToShoppingCart*

8.3.9.3 Order GSG

- Selling the largest quantities of products
    - Facilitate product selection to the customer
        - Customer browses and selects products
        - *ListCatalogs/ListProducts*

        - Add products to the shopping cart
        - Proceed in checking out | Continue shopping
        - *MoveItemsToOrder*
        - Show billing and shipping addresses
        - *ShowBillingAddress*
        - *ShowShippingAddress*
        - *VerifyBillShipProcess*
            - Verifying payment process
            - *VerifyPaymentProcess*
                - Process the order
                - *UpdateInventory*
- Making maximum number of orders to be shipped out
    - Placing orders by customer
    - *ProcessOrder*

8.3.9.4 Customer GSG

- Keep track of all customers
    - Customer profile
    - Customer Registration
    - Maintain Customer Profile
        - New Customer
        - *CreateCustomerRecord*
        - Update customer's data
            - Update Billing data
            - *UpdateBillingdata*
            - Update Shipping data
            - *UpdateShippingdata*
            - Update Payment data
            - *UpdatePaymentdata*
            - Update contact info
            - *Updatecontactinfo*
        - Delete customer's profile
        - *DeleteCustomerRecord*
        - Show customer's profile
        - *ShowCustomer*

8.3.10 Service Allocation

8.3.10.1 Customer Component Services

Service	Description
CreateCustomerRecord	Having a profile about each customer wants to use the web site
UpdateBillingdata *UpdateShippingdata* 8.3.10.1.1.1.1.1 UpdatePaymentdata	
Updatecontactinfo	Such as phone, email address, work, ... etc. Customer cannot update his USERID, but he can update the PASSWORD
DeleteCustomerRecord	Nullifying customer's record in the database
ShowCustomer	Show customer profile contents when he wants to update

8.3.10.2 Shopping Cart Component Services

Service	Description
ShowCatalog	List main catalogs that customer is able to choose what ever he wants in a user-friendly manner
ShowProduct / *ShowProductDetail*	After selecting a catalog, all products will be listed, and the customer is able either to see the detail of the selected product, or to add it to the shopping cart
AddProductToShoppingCart	Adding the selected product to the shopping cart
DeleteItem	Customer deletes an item from the shopping cart by entering (0) value in quantity text box and clicks update option
UpdateQuantities	Customer updates item's quantities by entering new value in quantity text box and clicks update option
UpdateShippingMethod	Customer can choose which shipping method he prefers, and then he has to click update option to make effects
SaveShoppingCart	Customer can save the current shopping cart as a draft in case he wants to postpone this transaction
GetCustomerLogin	Registered customers need to use their account, but new customer has to create an account

ShowBillingAddress/ ShowShippingAddress	Retrieve billing and shipping addresses from the database
SaveBillingAddress/ SaveShippingAddress	Save billing and shipping addresses in case of changes
ShowPaymentData	If it exists, otherwise retrieve null
SavePaymentData	In case of changes
SaveOrder	The order will be saved for tracking purpose
ShowOrderTransaction	List order items in a receipt that would be printed out by the customer
ShowPreferredItems	Wish List
ShowCustomerData	View customer's profile in case he wants to update or review
ShowShoppingCart	View the current shopping cart
One-click Shopping (*SubmitOrder*)	Submit the order in case billing, shipping, and payment data is already set for each customer
ComparePrice	Price Comparison for the same product with other merchants
RegisterCustomer	Keeping a profile about the customer in Ebazaar database
SavePurchaseHistory	Keeping track of the history of purchases for each customer
UpdatePurchase	Maintain number of purchases for each customer
ShowOtherPurchased	Show Customer what other purchasers have bought
ShowFavoriteItems	Show customer favorite items may come with the selected product

8.3.10.3 Product Catalog Components Services

Service	Description
ProduceProduct	Having a manufacture to do so
8.3.10.3.1.1.1.1.1 *FindSeller*	Locating the most valuable sellers
PurchaseProduct	Getting products from the seller
StoreNewQuantity	Changed product's quantity in the inventory
Maintain Catalog/ Maintain Product	Add | Delete | Update | View Catalog / Product
AddCatalog/AddProduct	
DeleteCatalog/ DeleteProduct	

8.3.10.3.1.1.1.1.2 UpdateCatalog/ *UpdateProduct*	
8.3.10.3.1.1.1.1.3 TransferCatalog/ TransferProduct	Move them to temporary store

8.3.10.4 Order Component Services

Service	Description
ListCatalogs/ListProducts	Customer browses and selects products
8.3.10.4.1.1.1.1.1 *MoveItemsToOrder*	Proceed in checking out
ShowBillingAddress/ ShowShippingAddress	Show the default billing and shipping data
SaveBillingAddress/ SaveShippingAddress	In case of changes
ShowPaymentData	
SavePaymentData	In case of changes
UpdateInventory	After processing the order, Ebazaar inventory should be deducted

8.3.11 Components Specification

For each of the components, we have a specification of their rules, services and attributes as defined below.

8.3.11.1 Order Components Specification

- Rules
  - Order cannot be processed before filling the shopping cart
  - Customers who place orders with a purchase amount >$100 and <$500 will get 10% off.
  - Customers who place orders with a purchase amount >$500 and <$900 will get 15% off.
  - Customers who place order with a purchase amount >$900 will get 20%
- Services
  - The same services mentioned in Service Allocation section
- Attributes
  - OrderNumber, OrderDate, OrderTotalAmount, OrderShippingMethod, ... etc

8.3.11.2 Product Catalog component Specification

- Rules
  - Ebazaar deals with any kind of products
  - Products should be categorized
- Services
  - The same services mentioned in Service Allocation section
- Attributes
  - CatalogID, CatalogName, ProductID, ProductName, Qty, UnitPrice, DateStored, ... etc

8.3.11.3 Customer Component Specification

- Rules
  - All Ebazaar e-business customers should be a registered in Ebazaar database.
  - Customers buy using credit cards
  - Ebazaar doesn't accept cash payments
- Services
  - The same services mentioned in Service Allocation section
- Attributes

- CustomerID, CustomerFName, CustomerLName, BillingAddress, ShippingAddress, PaymentData... etc

8.3.11.4 Shopping Cart Component Specification

- Rules
  - Shopping cart should be verified before it goes to order processing
- Services
  - The same services mentioned in Service Allocation section
- Attributes
  - SCID, CustomerID, ShippingAddress, BillingAddress, ... etc

8.3.12 Structuring Large-grained Enterprise Component

E. Component	Sub-Components
Customer	- ICustomerSubsystem - Customer - Custome SubsystemFaçade - DBCustomerRecord - Address
Shopping Cart	- IShoppingCartSubsystem - ShoppingCart - ShoppingCartSubsystemFacade - DBShoppingCart - DBShoppingCartTransaction - ShoppingCartDBConnection - ShoppingCartController
Order	- IOrderSubsystem - Order - OrderSC - OrderItems - OrderDB - OrderSubsystemFacade
Product Catalog	- ICatalogSubsystem - Catalog - Product - ProductItems - CatalogSubsystemFacade - DBCatalog - DBMediator - DBProduct

8.3.13 Technology Realization Mapping

The realization of the functionality of each of the components is now chosen and mapped to a physical realization.

Component	Services	Physical presentation
Customer	*CreateCustomerRecord*	DBCustomerRecord
	GetShippingdata	Customer
	GetBillingdata	Customer
	GetCustomer	Customer
	UpdateBillingdata	DBCustomerRecord
	UpdateShippingdata	DBCustomerRecord
	8.3.13.1.1.1.1.1 UpdatePaymentdata	DBCustomerRecord
	Updatecontactinfo	DBCustomerRecord
	DeleteCustomerRecord	DBCustomerRecord
	ShowCustomer	DBCustomerRecord
Shopping Cart	8.3.13.1.1.1.1.2 ShowCatalog	DBCatalog
	ShowProduct / *ShowProductDetail*	DBProduct DBProduct
	AddProductToShoppingCart	ShoppingCart
	DeleteItem	ShoppingCart
	UpdateQuantities	ShoppingCart
	UpdateShippingMethod	ShoppingCart
	SaveShoppingCart	DBShoppingCart
	GetCustomerLogin	Customer
	ShowBillingAddress/ *ShowShippingAddress*	Customer Customer
	SaveBillingAddress/ *SaveShippingAddress*	Customer Customer
	ShowPaymentData	Customer
	SavePaymentData	Customer
	SaveOrder	ShoppingCart
	ShowOrderTransaction	Order
	ShowPreferredItems	Order
	ShowCustomerData	Customer
	ShowShoppingCart	ShoppingCart
	One-click Shopping (*SubmitOrder*)	Order
	ComparePrice	Product (Ebazaar | Amazon)
	RegisterCustomer	Customer
	SavePurchaseHistory	Order
	UpdatePurchase	Order
	ShowOtherPurchased	Order

	ShowFavoriteItems	Order
Product Catalog	*ProduceProduct*	Product (Ebazaar | Amazon)
	8.3.13.1.1.1.1.1.3 *FindSeller*	Product (Ebazaar | Amazon)
	PurchaseProduct	Product (Ebazaar | Amazon)
	StoreNewQuantity	Product (Ebazaar | Amazon)
	AddCatalog/ AddProduct	Catalog (Ebazaar | Amazon) / Product (Ebazaar | Amazon)
	DeleteCatalog/ DeleteProduct	Catalog (Ebazaar | Amazon) / Product (Ebazaar | Amazon)
	8.3.13.1.1.1.1.1.4 UpdateCatalog/ *UpdateProduct*	Catalog (Ebazaar | Amazon) / Product (Ebazaar | Amazon)
	8.3.13.1.1.1.1.1.5 TransferCatalog/ TransferProduct	Catalog (Ebazaar | Amazon) / Product (Ebazaar | Amazon)
Order	*ListCatalogs/ ListProducts*	Catalog (Ebazaar | Amazon) / Product (Ebazaar | Amazon)
	8.3.13.1.1.1.1.1.6 *MoveItemsToOrder*	ShoppingCart
	ShowBillingAddress/ ShowShippingAddress	Customer Customer
	SaveBillingAddress/ SaveShippingAddress	Customer Customer
	ShowPaymentData	Customer
	SavePaymentData	Customer
	UpdateInventory	Order

8.3.14 E-bazaar Grammar

The following is an LL(1) grammar for the e-bazaar canonical application case study. This code is *executable* in the Business Compiler.

```
initial symbol = Start

Start = {#initTestData, Online Shopping, Epilogue}

Online Shopping = {Identification, Presentation}

Identification = {Challenge User with Login, Verify UserID and Password}

Challenge User with Login = {}

Verify UserID and Password ={}

Presentation = {Display Menu}

Display Menu ={Options}
Options = {
{customer, Customer}|
{administrator, Administrator}|
{stop, Epilogue}}

Customer = {
{online Purchase, Online Purchase}|
{orders History, Orders History}}

Online Purchase = {#msg(main catalog), {browse Items, Browse Items}|{main Menu, Close}}

Browse Items = {
#msg(catalog items),
{product Detail, Product Detail}|
{main Catalog List, Online Purchase}|
{main Menu, Close}}

Product Detail = {
#msg(product detail),
{add to Cart, Add to Cart}|
{continue Shopping, Online Purchase}}

Add to Cart = {
#msg(add to cart),
{main Catalog, Online Purchase}|
{proceed CheckOut, Proceed CheckOut}}

Proceed CheckOut = {
#msg(check out),
{payments, Payments}|
{main Catalog, Online Purchase}}

Payments = {
#msg(payment frame),
{terms and Conditions, Terms and Conditions}|
{main Catalog, Online Purchase}}
Terms and Conditions = {#msg(terms and conditions), Accept Terms and Conditions then
Proceed}
Accept Terms and Conditions then Proceed = {
{submit Order, Submit Order}|
{save Order, Save Order}|
{cancel, Presentation}}
```

```
Submit Order = {
{#msg(submit order)},
{continue Shopping, Online Purchase}|
{cancel, Presentation}}
Save Order = {#msg(save order), Presentation}

Orders History = {View Order History}

View Order History = {
#msg(view order history),
{view Order Detail, View Order Detail}|
{main Menu, Close}}
View Order Detail = {
#msg(order detail),
{view Items, View Items}|
{main Menu, Close}}
View Items = {#orderItems, Close}

Close = {Options}

Administrator = {
{maintain Product Catalog, Maintain Product Catalog}|
{maintain Product Types, Maintain Product Types}}

Maintain Product Catalog = {}
Maintain Product Types = {}

Epilogue ={#msg(Done)}
```

8.3.15 Dynamic Controller Architecture for E-Bazaar

The above grammar has been used to drive the dynamic controller indicated below:

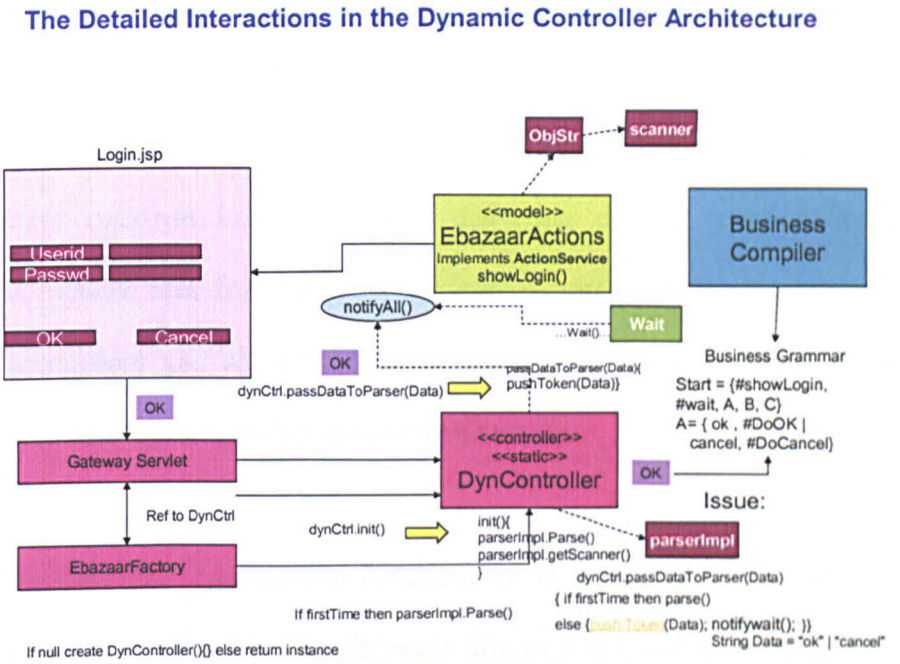


Figure 53: detailed Interactions of a Dynamic Controller Architecture

Figure 53 depicts the detailed interactions of a dynamic controller used in the e-bazaar case study.

9 Chapter Nine: Conclusion

9.1 The Problems and Their Solutions using GOOD

Building reliable and stable software edifices that withstand the test of changing requirements -- functional and non-functional-- has been found to be a complex, daunting and often unattainable task as many cancelled projects give testimony. This history of less-than-successful projects that have over-run budgets without delivering desired functionality within acceptable time frame and service level requirements is a testimony to the complexity and elusive nature of designing and maintaining stable and reliable, yet changeable software architectures.

Building and maintaining functionality in a software application is an expensive enterprise. To decrease this expense, software engineers have turned to rapid prototyping, to building software modules [77], components or assets that only need to be assembled. Thus, reusing units of available or easily obtainable assets has been the holy grail of system development. Its attainment would be synonymous with attaining business competitive advantage and by extension, the achievement of business objectives and success. This initiative is synonymous with the focus on component-based development.

Lack of reuse or fragility with respect to changing requirements is not the only cause of project failure. Organizational issues and politics, human factors, communications, expectation and risk management are among the other causes of a project's downfall. The above can be categorized as the organizational factors in project success.

The inability to meet the expectations and requirements of the business clients is one of the more salient aspects of failed projects where billions of dollars are essentially wasted and precious human time is misspent. Either, the wrong functionality was being built or the right one could not adapt itself to change fast enough.

Thus, the ability to integrate changing requirements into existing functionality is a key success criterion for project success. This criterion is adaptability or changeability.

Grammar-Oriented Object Design (GOOD) uses Domain-specific Languages (DSL) to provide a formal specification for the composition (static) and flow (dynamic) of a set of loosely coupled, dynamically-aggregated set of software components that support a business domain.

In this dissertation, GOOD is proposed as a software engineering method that facilitates the creation of a dynamically re-configurable architectural

style by providing a more seamless mechanism that maps business architecture onto component-based, service-oriented software architecture. This is achieved in such a fashion that changes to requirements are primarily controlled by the representation of the business architecture. The resulting architectural style can be used to represent the structure, composition and flow of software components providing services that are formally defined by a configurable profile that externalizes the variations of the semantics of a component services architecture in the form of *manners*. Manners can be implemented using domain-specific business languages (DSBL).

Further, the re-configurability and dynamic re-composition capabilities of software architecture are specified and reported on. The software architecture uses a spectrum of small to coarse-grained components and services for accomplishing this goal.

The unique approach outlined here is to use domain-specific languages, patterns and component-based and service-oriented software engineering to produce a highly re-configurable architectural style. In this approach, we do not advocate the usage of DSBLs and code generators to produce a domain-specific implementation of a running application. Rather, we apply a unique and well-formed combination of general-purpose programming languages

such as Java or C# with domain-specific languages (applicable to business domains) in order to define Configurable Profiles for Enterprise-scale components that expose services for composition and invocation.

This thesis also includes a review of the pertinent literature that has been published on component-based software engineering, domain-specific languages and software architecture and architectural styles. This establishes the framework in which GOOD is able to be effectively utilized.

Thus, dynamic re-configuration must be realized and implemented in a pervasive fashion across the following spectrum.

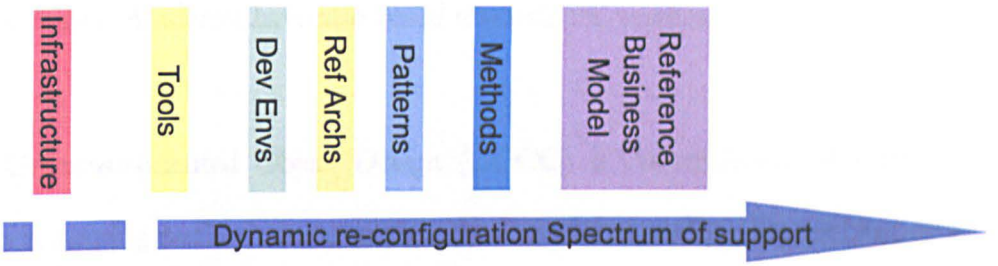


Figure 54: GOOD provides the capability to create pervasive dynamic reconfigurability across this spectrum

In order to re-configure a value net to achieve competitive advantage, an integrated architecture will enable responsiveness but not drive it. The driving factor is the capability to define configurations and then to dynamically alter them to meet new functionality and quality of service requirements. Virtualization is partly enabled through externalization of composition and

flow; responsiveness comes from the capability of dynamic re-configurability (DR). DR should be supported across the spectrum of our infrastructure, tools, development environments, reference architectures, best-practices and methods; all the way up to reference business models.

This thesis has demonstrated that Grammar-oriented Object design provides the capabilities needed to create the pervasive dynamically reconfigurability required.

9.2 PRIMARY CONTRIBUTIONS

GOOD has been tested and implemented on many industrial strength projects since 2001 and has been adopted as a method and is in use in industry. Academia have also based research and publications on GOOD.

Grammar-oriented Object Design (GOOD) is An evolution of software engineering methods for creating a dynamically re-configurable architectural style based on enterprise components and services. An integration of component-based, service-oriented engineering with domain-specific languages, software architecture, business rules, self-integration of context-aware autonomous components.

Business Grammars represent the flow, structure and composition of a business domain.

Variation-oriented Analysis and Design (VOA/VOD) are a set of 6 techniques and artifacts along with examples to partition a system or part thereof into its more stable and less stable aspects.

Subsystem Analysis identifies key subsystems within the business domain and provides a formal description for them.

Goal-oriented Component identification and Specification consists of the Identification and specification of candidate component abstractions during business analysis based on the notion of encapsulating design decisions.

Reuse levels consists of the levels of base class, aggregation, inheritance hierarchies, clusters, subsystems, frameworks, patterns, domain models. Each of these can be chosen as the unit of abstraction, and we are not constrained to use the object or class as the sole unit of abstraction.

Manners are the rules governing behavior of a component within a given context. They are presented with a set of patterns for its implementation. Context-aware Software Components define the characteristics necessary to design and implement dynamically reconfigurable architecture by understanding how to behave within a given context as defined by the manners that have been externalized in the components.

An Enterprise Component provides a standard and consistent way of designing large-grained components in enterprise content through the use of patterns.

CBDi Pattern Language is a Pattern Language for Component-based Development and Integration.

The Rule Pattern Language is a Pattern Language for Business Rules Modeling, Design and Implementation. It defines a spectrum of design and implementation options that can be used within a given context as appropriate.

Patterns for Web Services Architecture are a Pattern Language for the design of service-oriented architectures using web services technologies.

Dynamic Configuration, Collaboration are used to produce just-in-time integrating, on-demand application component assembly. They are the cornerstones of a dynamically reconfigurable architecture.

Mapping Business Architecture to Software Architecture consists of defining a set of steps and artifacts that result in the more seamless mapping of business architecture to component-based software architecture. This helps bridge the semantic gap between business and I/T through the creation of an abstract formal specification of the high-level business functionality that can be mapped directly onto component-base software architecture. Extensions to current methods for component-based and service-oriented software engineering

A set of activities and workproducts have been identified and utilized on projects since 1994 that extend object-oriented methods for component-based development.

Dynamically re-configurable architectural style is a blueprint (components, connectors and constraints) for creating an architecture that will support dynamic re-configuration and re-composition.

9.3 THE FUTURE

Future work is anticipated to gain even further industrial strength using the methods, architecture, tools and techniques of GOOD. This will be enhanced to provide the ability for software to be implemented as context-aware components on a large scale.

An industry around context-aware components will be created. Business modelers will create and retain business languages as key enterprise assets. Large integrators will produce generic business languages that will facilitate prototyping, design and implementation of software.

Work on formalizing the structure of the dynamics of changing requirements to help management information systems make decisions of software projects.

BIBLIOGRAPHY

[1] Arie van Deursen - Paul Klint - Joost Visser, Domain-Specific Languages: An Annotated Bibliography. http://www.cwi.nl/~arie/papers/dslbib/Sim96.

[2] Agrawal, R., R. Bayardo, D. Gruhl, and S. Papdimitriou. Vinci: A service-oriented architecture for rapid development of web applications. In WWW10, Hongkong, May 2001.

[3] Allen, B., and Holtzman, P., "Simplifying the Construction of Domain-Specific Automatic Programming Systems: The NASA Automated Software Development Workstation Project," Proceedings of the Space Operations Automation and Robotics Workshop, (Houston, TX).

[4] Arango, G., Domain analysis: from art form to engineering discipline; Proceedings of the fifth international workshop on Software specification and design, 1989, Pages 152 – 159.

[5] Arango, G., Software Reusability, chapter 2. Domain analysis methods, pages 17-49. Workshops M.E. Horwood, London 1994.

[6] Arango Thibault, S. A. , Marlet, R., and Consel, C., Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation, IEEE Transactions on Software Engineering, vol. 25, pp. 363-377, May 1999.

[7] Ardis, M., Dudak, P., Dor, L., Leu, W., Nakatani, L., Olsen, and Pontrelli, P., "Domain Engineered Configuration Control", in Proceedings of the First Software Product Line Conference (P. Donohoe, ed.), pp. 479-493, Aug.2000.

[8] Arsanjani, A., "Analysis and Design of Distributed Java Business Frameworks using Domain Patterns", Proceedings of TOOLS '99,IEEE Computer Society Press 1999, pp. 490-500.

[9] Arsanjani, A., "CBDi : A Pattern Language for Component-based Development and Integration", European Conference on Pattern Languages of Programming, 2001.

[10] Arsanjani, A., "Dynamic Configuration and Collaboration of Components with Self -description," position paper submitted to ECOOP2001 Workshop on Active Object Models and Meta-modeling, 2001.

[11] Arsanjani, A., "GOOD: Grammar-oriented Object design", Position Paper for OOPSLA Workshop on Metadata and Active Object Models, 1998,Vancouver, British Columbia.

[12] Arsanjani, A., Alpigini., J., "Using Grammar-oriented Object Design to Seamlessly Map Business Models to Software Architectures", Proceedings of the IASTED 2001 conference on Modeling and Simulation, Pittsburgh, PA,2001.

[13] Arsanjani, A., Rule Object: A Pattern Language for Flexible Modeling and Construction of Business Rules, Washington University Technical Report number: wucs-00-29, Proceedings of the Pattern Languages of Program Design, 2000.

[14] Arsanjani, A., Alpigini, J., Zedan, H., Externalization of Component Manners to Achieve a Highly re-configurable Architectural Style, proceedings of the IEEE ICSM 2002 conference.

[15] Arsanjani, A., Enterprise Component: A Compound Pattern for Building Component Architectures, Proceedings of TOOLS 2001.

[16] Arsanjani, A., Grammar-oriented Object Design: Creating adaptive collaborations and dynamic configurations with self-describing components and services, Proceedings of TOOLS 2001.

[17] Arsanjani, A., Patterns of Symmetry and Stability in Software Architecture, Pattern Languages of Programming 2001.

[18] Arsanjani, A., "Enterprise Components and Services", Communications of the ACM, Oct 2002.

[19] Levi, K., Arsanjani, A.,"A Goal-oriented Approach to Component Identification and Specification", Communications of the ACM, Oct 2002.

[20] Batory, D., Coglianese, L., Goodwin, M., and Smith, R. ,"A Domain Model for Avionics Software," Tech. Rep. ADAGE-UT-92-01, Department of Computer Science, University of Texas, Austin, Texas, Feb 1992.

[21] Boehm, Barry, Software Engineering Economics, Addison-Wesley 1984.

[22] Booch, G., Rumbaugh, J., Jacobson, I., Unified Modeling Language User Guide, The, The Addison-Wesley Object Technology Series, 1999.

[23] Bory, Deimel, Henn, Koskimies, Pasli, Pomberger, Pree, Stal and Syperski, "What characterizes a software component?", Software - Concept and Tools (1998),Vol 19, No.1, pp. 48-56.

[24] Bosch, Jan, Design and Use of Software Architectures, Adopting and Evolving a Product-Line Approach, ISBN 0-201-67494-7, May 2000.

[25] Szyperski, C., Component software, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1998 .

[26] Clements, Paul; Northrop, Linda; Building Product Line Architectures, Prentice-Hall 2001.

[27] Coplien, J.O. and Schmidt, D.C. Pattern Languages of Program Design, Addison-Wesley, Reading, MA, 1995.

[28] Gacek, C., Anastasopoules, M., Implementing product line variabilities, ACM SIGSOFT Software Engineering Notes ,proceedings of SSR '01 on 2001 symposium on software reusability : puttings software reuse in context May 2001, Volume 26.

[29] Dikel, D., Kane, D., Ornburn, S., Loftus, W., Wilson, J., 'Applying Software Product-Line Architecture,' IEEE Computer, pp. 49-55,August 1997.

[30] D'Souza, D., Wills, A.C., Objects, Frameworks and Components with UML, Addison-Wesley 1998.

[31] Daniels, J., Cheesman, J., UML Components, Addison-Wesley, 2000, pp. 68-73.

[32] Johnson, R., Foote., B., Designing Reusable Classes, Journal of Object-oriented Programming, 1996.

[33] Roberts, D., Johnson, R., "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", Proceedings of Pattern Languages of Programs, Allerton Park, Illinois, September 1996.

[34] Enterprise JavaBeans Specification: ttp://java.sun.com/products/ejb/2.0.html

[35] Erikson, Hans-Erik; Penker, Magnus; Business Modeling with UML, John Wiley and Sons, 2000, pp. 66-75,pp. 370-371.

[36] Beck, K., Extreme Programming Explained, Addison-Wesley, 2000.

[37] Fayad, M., "Stability in Software Architecture", CACM 2001.

[38] Bachmann, Felix; Bass, Len; Managing variability in software architectures, ACM SIGSOFT Software Engineering Notes ,proceedings of SSR '01 on

2001 symposium on software reusability : putting software reuse in context May 2001,Volume 26 Issue 3.

[39] Fowler, M., Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.

[40] Frank Buschmann , Regine Meunier , Hans Rohnert , Peter Sommerlad , Michael Stal, Pattern-oriented software architecture, John Wiley & Sons, Inc., New York, NY, 1996.

[41] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, 'Aspect-Oriented Programming,' Proceedings of ECOOP'97 (invited paper), pp. 220-242, LNCS 1241,1997.

[42] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of reusable Object-oriented Software, Addison-Wesley, 1994.

[43] Garlan, D., Kompanek, A., Melton R., and Monroe R., Architectural Style: An Object-Oriented Approach, February, 1996 at http://www.cs.cmu.edu/afs/ cs/project/able/www/able/papers_bib.html.

[44] Gerald C.G., Lutz, R., An approach to architectural analysis of product lines, Proceedings of the 22nd International Conference on Software Engineering June 2000.

[45] Graham I., Migrating to Object Technology, Addison-Wesley, Wokingham UK, 1995.

[46] Heineman, George, Component-based Software Engineering, Addison-Wesley, 2001.

[47] IBM Object Technology Center, Developing Object-oriented Software: An Experience-based Approach, Prentice-Hall, 1997, pp. 192-232.

[48] Jacobson, Ivar, Griss, Martin, Jonsson, Patrik, Software reuse, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1997 .

[49] Bosch, J. "Design of an Object-Oriented Framework for Measurement Systems," Object-Oriented Application Frameworks, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, 1998. (forthcoming), March 1998.

[50] Bosch, J., "Design Patterns as Language Constructs", Journal of Object-Oriented Programming, Vol. 11, No. 2, pp. 18-32, May 1998.

[51] Bosch, J., Molin, P., 'Software Architecture Design: Evaluation and Transformation,' Proceedings ECBS'99 (forthcoming), March1999.

[52] Ramming, J. C., ed. Proceedings of the USENIX Conference on Domain-Specific Languages, Berkeley, CA, October 15-17 1997. USENIX Association.

[53] Bentley, J.L., Programming pearls: Little languages, Communications of the ACM, 29(8):711-721, August 1986.

[54] Jacobson, I., Ericsson, M. and Jacobson, A. The Object Advantage: business process engineering with object technology, Addison-Wesley, 1995.

[55] Jacobson I., Christerson M., Jonsson P. and Övergaard G. (1992) Object-Oriented Software Engineering: a use case driven approach, Addison-Wesley.

[56] Bosch, J., Object acquaintance selection and binding, Theory and Practice of Object Systems, v.4 n.3, p.151-168, Aug. 10, 1998.

[57] Bosch, J., Product-line architectures in industry, Proceedings of the 1999 international conference on Software engineering May 1999.

[58] Jazayeri, M., Product Lines, Springer-Verlag, 1992.

[59] Bayer , J., Girard , Jean-François, Würthner , Martin, DeBaud , Jean-Marc, Apel, Martin, Transitioning legacy assets to a product line architecture, ACM SIGSOFT Software Engineering Notes , Proceedings of the 7thEuropean Engineering Conference.

[60] Savolainen , Juha; Kuusela, Juha; Volatility analysis framework for product lines, ACM SIGSOFT Software Engineering Notes ,proceedings of SSR '01 on 2001 symposium on software reusability : putting software reuse in context May 2001, Volume 26 Issue 3.

[61] Kevin J. Sullivan , William G.Griswold , Yuanfang Cai , Ben Hallen The structure and value of modularity in software design, ACM SIGSOFT Software Engineering Notes , Proceedings of the 8th European software engineering conference held jointly with 9th ACM SI

[62] Yagoub, Khaled; Florescu, Daniela; Issarny, Valerie; Valduriez, Patrick. Caching Strategies for Data-Intensive Web Sites. In Proceedings of the 24th International Conference on Very Large Databases (VLDB), September 2000, Cairo, Egypt.

[63] Bass, L., Clements, P., Cohen, S., Northrop, L., Withey, J., 'Product Line Practice Workshop Report, Technical Report CMUSEZ-97-TR-003, Software Engineering Institute, June 1997.

[64] Hassani, M., Stewart, D., A Mechanism for communicating in Dynamically reconfigurable Embedded Systems, Proc. Of High Assurance Software Engineering, 1997.

[65] Simos, M.A.,'Lateral Domains: Beyond Product-Line Thinking,' Proceedings Workshop on Institutionalizing Software Reuse (WISR-8), 1997.

[66] Meyer, B. Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, NJ, 1997

[67] Mattsson, M., 'Object-Oriented Frameworks - a survey of methodological issues', Licentiate thesis, Department of Computer Science, Lund University, 1996,

[68] Shaw , M., Garlan, D., Software architecture, Prentice-Hall, Inc., Upper SaddleRiver, NJ, 1996.

[69] McDavid, D., A Standard for Business Architecture Description, IBM Systems Journal. Vol 38, No. 1, 1999.

[70] Object Constraint Language, Addison-Wesley, 1998.

[71] Kruchten, P., "The 4+1 View Model of Architecture," IEEE Software, pp. 42-50, November 1995.

[72] Parnas, D.L., "On the Criteria to used in Decomposing Systems into Modules", CACM 1972.

[73] PerOlof Bengtsson, Nico Lassing, Jan Boschand Hans van Vliet, Analyzing Software Architectures for Modifiability, (HK/R Research Report 2000:11, ISSN: 1103-1581.

[74] Coad, Peter, Object Modeling in Color, Prentice-Hall, 2000.

[75] Pree, W., Patterns and Hotspots. Prentice-hall 1993.

[76] Proceedings of the second USENIX Conference on Domain-Specific Languages. USENIX Association, October 3-5 1999.

[77] Johnson, R., Foote, B., 'Designing Reusable Classes, Journal of Object-Oriented Programming, Vol. 1 (2), pp, 22-25, 1988.

[78] Macala, R., Stuckey, L.D. , Gross, D.C., 'Managing Domain-Specific Product-Line Development,' IEEE Software, pp. 57-67, 1996.

[79], Opdyke, William. Refactoring University of Illinios at Urbana-Champaign PhD Thesis, 1996.

[80] Fowler, M., Beck, K. (Contributor), Brant, J. (Contributor), Opdyke, W., Roberts, D., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2001.

[81] Lajoie, R. and Keller, Rudolf K., 'Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert,"Object-Oriented Technology for Database and Software Systems", V.S. Alagar and R. Missaoui (eds), World Scientific Publishing.

[82] Rumbaugh, J., Jacobson, I., Booch, G., Unified Software Development Process, The Addison-Wesley Object Technology Series, 1999.

[83] Rumbaugh, J., et al., Object Modeling Technique, Prentice Hall, 1991.

[84] Rumbaugh, Booch, Jacobson, The Unified Modeling Language Reference Manual, The Addison-Wesley Object Technology Series, 1999.

[85] Kamin, S. ed. DSL '97 - First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL '97, Paris, France, January 1997. University of Illinois Computer Science Report.

[86] Shlaer, S. Mellor, S.J., 'Recursive Design of an Application-Independent Architecture,' IEEE Software, pp. 61- 72,January/February 1997.

[87] Ted Lewis, Object oriented application frameworks, Manning Publications Co., Greenwich, CT, 1995 .

[88] Thibault, S. A. , Marlet, R., and Consel, C., Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation, IEEE Transactions on Software Engineering, vol. 25, pp. 363-377, May 1999.

[89] Issarny, V., Bellissard, L., Riveill, M., Zarras, A. Component-based Programming of Distributed Applications. In Recent Advances in Distributed

Systems. S. Krakowiak and S. Shrivastava editors. Springer Verlag, LNCS 1752. 2000.

[90] Issarny V., Banatre, M., Charpiot B., Menaud, J. Quality of Service and Electronic Newspaper: The Etel Solution. In Recent Advances in Distributed Systems. S. Krakowiak and S.Shrivastava editors. Springer Verlag, LNCS 1752. 2000.

[91] Vlissides, J.M., Coplien, J.O. and Kerth, N.L. ed. Pattern Languages of Program Design 2, Addison-Wesley,1996.

[92] Wiederhold, Wegner, Ceri: "Towards Megaprogramming"; CACM, June 1992.

[93] Tracz, W., Product-line architectures, Proceedings of the fifth symposium on Software reusability May 1999.

[94] Requirements Documents for Web Services Architecture and Description, http://xml.coverpages.org/ni2002-04-29-a.html.

[95] Widen, T.. Formal language design in the context of domain engineering. Master 's thesis, Oregon Graduate Institute, June 1998.

[96] Arie van Deursen. Executable Language Definitions. Ilic dissertation series, CWI, P.O. Box, 94079, 1090 GB Amsterdam, 1994.

[97] Florescu, D., Levy A.W., and Suciu, D., and Yagoub, K., Optimization of Run-time Management of Data Intensive Web-sites, The {VLDB} Journal, pp. 627-638,1999.

[98] Tarr, P., Ossher, H. Harrison, W., Sutton, S. Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, International Conference on Software Engineering, 1999.

[99] Spinellis, D., Notable design patterns for domain specific languages. Journal of Systems and Software, 56(1):91-99, February 2001. http://softlab.icsd.aegean.gr/~dspin/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html

[100] Joseph W. Yoder & Ralph Johnson. "***The Adaptive Object Model Architectural Style***" Published in The Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02) at the World Computer Congress in Montreal 2002, August 2002. Software Architecture System Design, Development and Maintenance Edited by Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela; Kluwer Academic Publishers 2002.

[101] Charles Herring, Adaptable and Adaptive Systems: The Intelligent Control Paradigm for Software Architecture, Working Conference on Complex and Dynamic Systems Architectures, December 12-14, 2001, Brisbane, Australia.

[102] Arsanjani, A.,Using Grammar-oriented Object Design to define Dynamic Configuration and Adaptive Collaboration of Proceedings of the International Conference on Software Maintenance (ICSM™02).

[103] Allen, B., and Holtzman, P., "Simplifying the Construction of Domain-Specific Automatic Programming Systems: The NASA Automated Software Development Workstation Project," *Proceedings of the Space Operations Automation and Robotics Workshop*, (Houston, TX), pp. 407-410, NASA Johnson Space Center, Aug. 1987.

[104] Ardis, M., Dudak, P., Dor, L., Leu, W., Nakatani, L., Olsen, and Pontrelli, P.,"Domain Engineered Configuration Control", in *Proceedings of*

the First Software Product Line Conference (P. Donohoe, ed.), pp. 479-493, Aug. 2000

[105] Batory, D., Coglianese, L., Goodwin, M., and Smith, R. , "A Domain Model for Avionics Software," Tech. Rep. ADAGE-UT-92-01, Department of Computer Science, University of Texas, Austin, Texas, Feb 1992.

[106] Michael Jackson. Specializing in software engineering. *IEEE Software*, 16(6):119-121, Nov/Dec 1999.

[107] Stephen C. Johnson and Michael E. Lesk. Language development tools. *Bell System Technical Journal*, 56(6):2155-2176, July-August 1987.[1] http://www.cwi.nl/~arie/papers/dslbib/

[108] J. C. Cleaveland. Building application generators. *IEEE Software*, pages 25-33, July 1988.

[109] Allen, P., *Realizing e-business with Components*, Addison-Wesley, 2000, pp. 64-71.

[110] Arsanjani, A., "A Domain-language Approach to Designing Dynamic Enterprise Component-based Architectures to Support Business Services," *Proceedings of TOOLS* 2001, IEEE Press, 2001.

[111] Business Extensions for UML, OMG . www.omg.org/uml.

[112] Krutchen, P., The Rational Unified Process: An Introduction, Addison-Wesley, 1999.

[113] Rumbaugh, J., Booch, G., Jacobson, I., *The Unified Modeling Language Reference*, Addison-Wesley, 1999.

[114] IBM Object Technology Center, *Developing Object-oriented Software: An Experience-based Approach*, Prentice-Hall, 1997, pp. 192-232.

[115] Marshall, C., *Enterprise Modeling*, Addison-Wesley, 2000, pp. 7-26.

[116] OASIS standards body, www.oasis.org

[117] Arsanjani, A., "Analysis and Design of Distributed Java Business Frameworks using Domain Patterns", *Proceedings of TOOLS '99*, IEEE Computer Society Press 1999, pp. 490-500.

[118] Evolving Frameworks A Pattern Language for Developing Object-Oriented Frameworks, Don Roberts, Ralph Johnson, University of Illinois.

[119] *Formal Specification and Documentation using Z: A Case Study Approach*, Jonathan Bowen. International Thomson Computer Press, International Thomson Publishing, 1996.

[120] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175-188. Copyright Kluwer, 1999.

[121] Kemmerer, Richard A., Integrating Formal Methods into the Development Process, September/October 1990 (Vol. 7, No. 5), pp. 37-50.

[122] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138-150, October 1968.

[123] Brad J. Cox. *Object-Oriented Programming, An evolutionary approach*. Addison-Wesley, 1986.

[124] Herzum, P., Business Component Factory, Addison-Wesley 2000.

[125] Szyperski, C., Component software, ACMPress/Addison-Wesley Publishing Co., New York, NY, 1998 .

[126] Model-View-Controller Pattern, http://www.enode.com/x/markup/tutorial/mvc.html .

[127] Arsanjani, A., Rule Object: A Pattern Language for Flexible Modeling and Construction of Business Rules, Washington University Technical Report number: wucs-00-29, Proceedings of the Pattern Languages of Program Design, 2000.

[128] Daniels, J., Cheesman, J., *UML Components*, Addison-Wesley, 2000, pp. 68-73.

[129] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999, pp.98-106, pp.122-130.

[130] Allen, R. (2002). *Workflow: An introduction.* The Workflow Management Coalition, http://www.wfmc.org/standards/docs/Workflow_An_Introduction.pdf.

[131] Michael VanHilst , David Notkin, Decoupling change from design, Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, p.58-69, October 16-18, 1996, San Francisco, California, United States

[132] Richard Helm , Ian M. Holland , Dipayan Gangopadhyay, Contracts: specifying behavioral compositions in object-oriented systems, ACM SIGPLAN Notices, v.25 n.10, p.169-180, Oct. 1990

[133] Workflow Management Coaliti vcbnm mnbv vn ,cv mon, http://www.wfmc.org/standards/docs/tc003v11.pdf

[134] Autonomic Computing, creating self-managing computing systems, http://www-3.ibm.com/autonomic/pdfs/ACwpFinal.pdf

[135] Fred Douglis, Ian Foster, Grid computing Comes of Age, IEEE Internet Computing July/August 2003.

[136] Sven Graupner, Vadim Kotov, Artur Andrzejak, Holger Trink, Service-Centric Globally Distributed Computing, IEEE Internet Computing July/August 2003.

[137] Thomas Schael, Workflow Management Systems for Process Organisations, Springer-Verlag, 1998.

[138] F. DeRemer and H. Kron, "*Programming in the Large versus Programming in the Small*," IEEE Transactions on Software Engineering, 2(2), pp. 80-87, 1976.

[139] Micro-Workflow: A Workflow Architecture Supporting, Compositional Object-Oriented Software Development, Ph.D. thesis, Computer Science Technical Report UIUCDCS-R-2000-2186, University of Illinois at Urbana-Champaign, October 2000, Urbana, Illinois

[140] P. Naur and B. Randell, (Ed.). Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, Brussels, Scientific Affairs Division, NATO, January 1969, 231 p.

[141] *Designing Enterprise Applications with the J2EE™ Platform, Second Edition,* Inderjeet Singh, Beth Stearns, Mark Johnson, Addison-Wesley, 2002.

[142] A study of technologies supporting development of software with an N-tier architecture, Samuel Aslund, 2001.

[143] Enterprise Application Integration Strategies for the Telecommunication Market, Martin Rosendahl, Erik Rune, 2000.

[144] The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. I. Foster, C. Kesselman, J.

Nick, S. Tuecke, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.